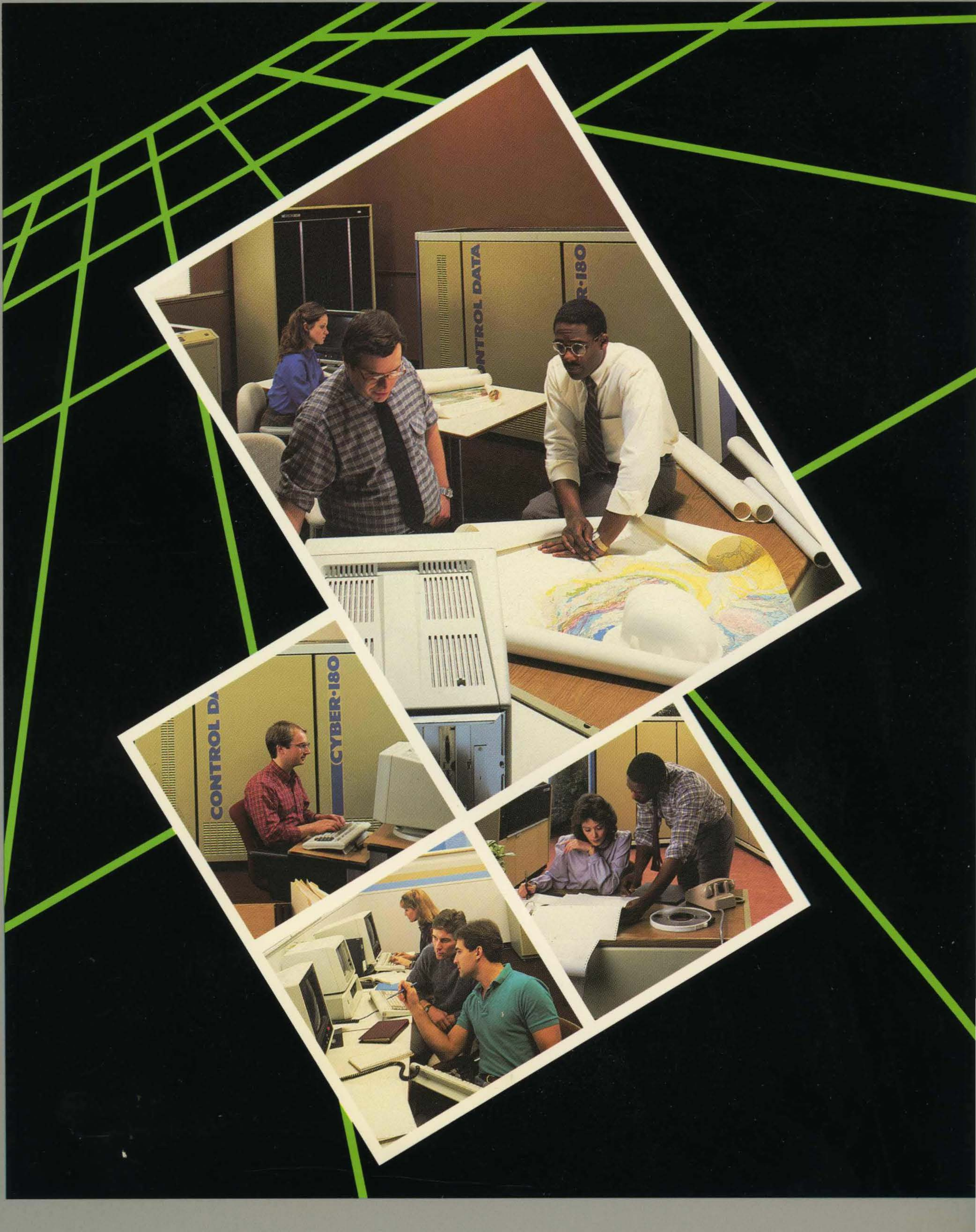
NOS/VE Build Utility

CONTROL DATA



Usage

60487413

NOS/VE
Build Utility

Usage

Publication Number 60487413

Manual History

Revision	System Version/ PSR Level	Product Version	Date
A	1.4.2/727	1.0	June 1989

Revision A of the Build Utility usage manual documents the Build Utility for NOS/VE at release level 1.4.2 in June 1989.

Contents

About This Manual

This manual documents the NOS/VE Build Utility. It describes how to set up an input file and use the Build Utility. It also documents the Build Utility subcommands and functions.

This manual is structured as follows:

- Chapters 1 through 4 describe how to create an input file and use the Build Utility.
- Chapter 5 is a reference section that describes the Build Utility subcommands and functions.
- Chapter 6 is an example section that provides three examples of Build Utility usage.
- Appendix A is a glossary of the Build Utility terms.
- Appendix B describes the NOS/VE manual set.
- Appendix C describes the Build Utility diagnostic messages.
- Appendix D describes some of the Build Utility concepts.

Audience

This manual is written primarily for developers and documentors of NOS/VE applications. Many of these developers will use the Build Utility from the Professional Programming Environment (PPE).

This manual assumes you are familiar with the following:

- NOS/VE System Command Language (SCL)
- Source Code Utility (SCU)
- EDIT_FILE Utility (EDIF)
- File system concepts

Conventions

All references to commands, subcommands, and functions in this manual use the full command name. Most commands have abbreviated forms, which you may find more convenient to use. All forms of every command are included in the individual command descriptions, which are located in chapter 5, Commands and Functions.

In addition, the following conventions apply throughout this manual.

Boldface	In a format, boldface type represents names and required parameters.
Italics	In a format, italic type represents optional parameters.
UPPERCASE	In a format, uppercase letters represent reserved words defined by the system for specific purposes. You must type these words exactly as shown.
lowercase	In a format, lowercase letters represent values you choose.
Shaded text	In examples of interactive terminal sessions, shaded text represents user input.
Numbers	All numbers are decimal.

The Build Utility makes no distinction between uppercase and lowercase letters. Names of commands, subcommands, functions, files and decks may be in uppercase or lowercase letters. For example, MY_FILE, my_file, and My_File are the same.

Submitting Comments

The last page of this manual is a comment sheet. Please tell us about any errors you found in this manual and any problems you had using it.

If the comment sheet in this manual has been used, please send your comments to us at this address:

Control Data Corporation
Technical Publications
P.O. Box 3492
Sunnyvale, California 94088-3492

Please include this information with your comments:

- The manual title and publication number (for this manual, specify, NOS/VE Build Utility Usage, 60487413) and the revision letter from the page footer.
- Your system's PSR level (if you know it).
- Your name, your company's name and address, your work phone number, and whether you want a reply.

Also, if you have access to SOLVER, the Control Data online facility for reporting problems, you can use it to submit comments about this manual. When it prompts you for a product identifier for your report, please specify BU8.

In Case You Need Assistance

Control Data's CYBER Software Support maintains a hotline to assist you if you have trouble using our products. If you need help beyond that provided in the documentation or find that the product does not perform as described, call us at one of the following numbers and a support analyst will work with you.

From the USA and Canada: (800) 345-9903

From other countries: (612) 851-4131

The preceding numbers are for help on product usage. Address questions about the physical packaging and/or distribution of printed manuals to Literature and Distribution Services at the following address:

Control Data Corporation
Literature and Distribution Services
308 North Dale Street
St. Paul, Minnesota 55103

or you can call (612) 292-2101. If you are a Control Data employee, call CONTROLNET® 243-2100 or (612) 292-2100.

Introduction 1

Introduction 1

The Build Utility is a NOS/VE command utility designed to simplify the updating of a file system or library. When the source text of a file system changes, the Build Utility updates only the sections of object code affected by the change. The Build Utility uses a user-defined description of the files or decks that comprise the system to determine whether the system is out of date.

If you are familiar with the UNIX(TM)[1] operating system, the Build Utility is similar to the UNIX *make* command.

The Build Utility is designed primarily for developers who design large software systems on NOS/VE. However, it can be used for any size system. The Build Utility does not set any limits on the size or complexity of a system beyond the limits set by NOS/VE or your site.

The Build Utility is easiest to run from within the Professional Programming Environment (PPE) because PPE automatically maintains the descriptions of your library system. However, the Build Utility can be used at the NOS/VE command level as well.

This chapter serves as an introduction to the Build Utility. It defines the Build Utility terminology and briefly describes how the Build Utility works.

1. UNIX is a trademark of AT&T Bell Laboratories.

Terminology

You will need to understand the following terms before learning the Build Utility. The definitions of these terms apply throughout this manual.

Build

The process by which the Build Utility constructs build targets. This process involves the executing of a transformation on the files that comprise a build target.

Build Target

A file that is the result of a transformation.

Data Dependency

The relationship between the outputs and inputs of a transformation. The contents of the output are dependent on the contents of the input. For example, the file produced by binding an object library is dependent on the object library.

Dependency Graph

A visual representation of data dependency between files.

File System

A collection of related files. In this manual, these files are used to construct a build target.

Input File

This file contains the information needed by the Build Utility. It is a required parameter of the BUILD_SOFTWARE command.

Source Library

An SCU file that consists of a collection of decks. Each deck has a header, which describes the deck.

Transformation

A sequence of one or more NOS/VE commands that manipulate data. For example, a call to the COBOL compiler is a transformation that changes COBOL source text to object code.

NOTE

If you require additional information, please refer to the appendix section of this manual. Appendix A contains a glossary of Build Utility and related terms. Appendix D provides a more detailed discussion of some of the above terms.

Without the Build Utility

Updating a file system or library can be a time-consuming process without the benefit of a tool like the Build Utility.

Without the Build Utility, updating a file system is often a tedious and time-consuming process. For example, suppose you write a COBOL program. You maintain a copy of the source code and the compiled object code. If you make a change to the source code without recompiling, then the object code is not current. In this case, you can either:

- Compile only the modules affected by the change.
- Compile the entire program.

For a small program, either of the above options is acceptable. Because the program is small, it is easy to identify which modules are affected by a change. Also, small programs compile quickly. So, you can recompile the entire program without wasting much time. In either case, the advantages of the Build Utility are not apparent.

For a large program, however, the process is not as simple. Identifying all the modules that need to be recompiled requires that you follow all file or deck references. This can be an enormous task. The other option is to perform a full build. A large program can take several hours to compile. Much of this time is wasted if only a small section of code is changed. The Build Utility was developed to minimize the time required to update a file system or library.

How the Build Utility Works

The Build Utility performs three steps: reads the Build Utility input file, analyzes the build target, and performs the assigned transformation. These three steps can be divided into two phases: data definition and data analysis.

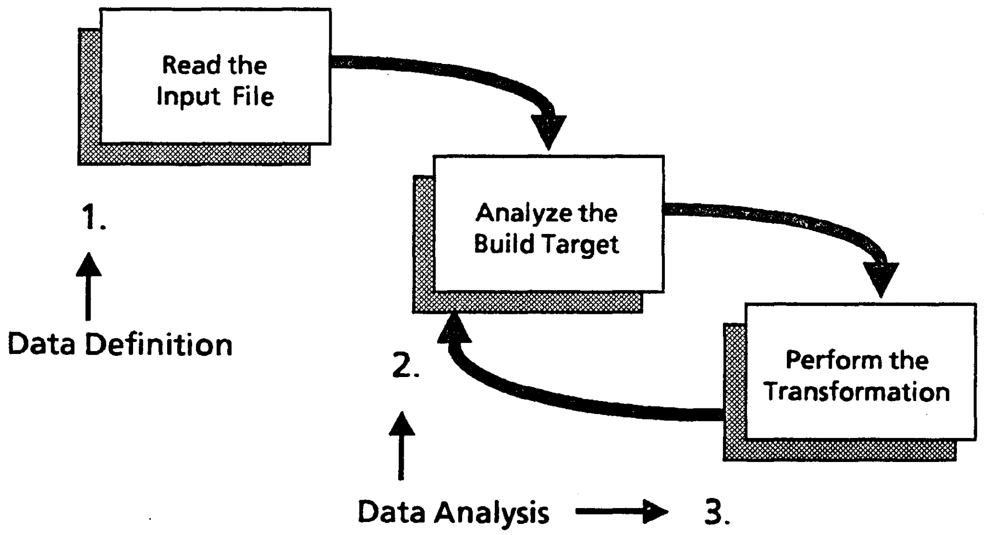
Data Definition

First, the Build Utility reads the input file. This file contains all the user-defined information about the system. The input file is described in detail in chapter 2, Using the Build Utility. This is called the data definition phase. During this phase, the Build Utility identifies the build targets and the decks or files that form the targets. Also during this phase, the transformations for each target are defined.

Data Analysis

Next, the Build Utility enters the data analysis phase. Each deck, file, or object module has a date/time stamp associated with it. This stamp indicates the last time the file or deck was changed. If the date/time stamp does not exist, the Build Utility assigns one. The Build Utility then compares the date/time stamp of the build target with the date/time stamp of the decks or files on which the target depends. If a deck or file has a more current date/time stamp, then the corresponding build target is considered out of date. The Build Utility performs the transformation and the build target becomes current. The Build Utility continues to identify the changed source and rebuild until there are no more build targets.

The diagram below summarizes the steps taken by the Build Utility.



For PPE Users

PPE users are a special case when using the Build Utility because PPE automatically maintains all the necessary information about a library system.

The Build Utility is designed primarily for developers using PPE to write NOS/VE applications. Using the Build Utility from PPE can be very simple. PPE sets up a default dependency condition. Therefore, you do not need to provide an input file when running the Build Utility from PPE. However, you are permitted to reference an input file if your system requires a more specialized definition than the PPE default. If you are familiar with the BUILD_DECKS and BUILD_CHANGED_DECKS commands in PPE and do not anticipate changing the default values of the build process, then you may want to skip to chapter 3, The Build Utility and PPE.

Is the Build Utility just a command interface for the PPE build commands?

No, the Build Utility is not just a command interface for the PPE build commands. The Build Utility can be used to update files outside of PPE. The PPE commands BUILD_DECKS and BUILD_CHANGED_DECKS can only be used on a library system within PPE.

In addition, the Build Utility increases the flexibility of the PPE build commands. Prior to the Build Utility, there were no parameters associated with either BUILD_DECKS or BUILD_CHANGED_DECKS. PPE used a default dependency for every system, and it was not possible to change it. Both commands now have a full set of parameters that allow you to redefine PPE's default conditions.

Using the Build Utility 2

Using the Build Utility 2

This chapter describes how to start the Build Utility, how to set up an input file, and the components of an input file. This chapter is designed for those users who are creating their own input file because they are using the Build Utility from NOS/VE. This chapter also provides a reference for PPE users who want to create their own input file to override the PPE defaults.

Starting the Build Utility

There are three ways to start the Build Utility:

- From NOS/VE, enter the BUILD_SOFTWARE command.
- From PPE, press the function key labeled BUILD or the function key labeled BUID.
- From PPE, enter the BUILD_DECKS or BUILD_CHANGED_DECKS command at the home line.

If you are using the Build Utility from NOS/VE, start the utility by entering the BUILD_SOFTWARE command at the NOS/VE command prompt. You must also pass the input file as a required parameter of the BUILD_SOFTWARE command. The following example shows a call to the Build Utility where the INPUT parameter is the name of the input file, called INFILE.

```
/build_software input=infile
```

or abbreviated,

```
/buis infile
```

For a complete description of all the parameters for the BUILD_SOFTWARE command, see chapter 5, Commands and Functions.

When using PPE there are two ways to initiate the Build Utility. You can press the function key labeled BUILD or the function key labeled BUID. The BUILD function causes the Build Utility to build the decks that have changed. The BUID function causes the Build Utility to build the decks that you have marked.

You can also enter the commands directly from the home line in PPE. Entering BUILD_DECKS is equivalent to using the BUID function key. Entering the command BUILD_CHANGED_DECKS is equivalent to using the BUILD function key.

When you start the Build Utility from PPE, you do not need to provide any parameters. PPE automatically maintains a default description of your system. However, it is possible to change these default settings provided by PPE. For information on declaring your own input file in PPE, see chapter 3, The Build Utility and PPE.

Creating the Input File

The input file is required for the BUILD_SOFTWARE command. You can create an input file by using the EDIT_FILE utility. The input file cannot be a deck in a source library.

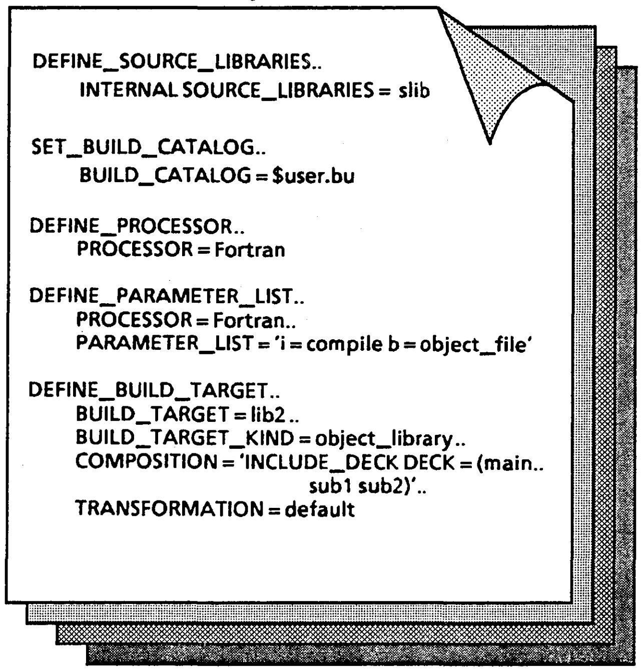
Before issuing the BUILD_SOFTWARE command from NOS/VE, you must create an input file. You can optionally specify an input file within PPE. The input file must be a file and cannot be a deck in a library. However, it can be text extracted from a deck to a file. You must specify the full path for the input file, or it must reside on the current working catalog.

An input file is simply an ASCII text file that contains the subcommands and functions as well as NOS/VE commands. It defines the following information:

- The data dependency in the system.
- The build target(s).
- The processors used in the transformation of the build target(s).
- The parameter list for the defined processors (optional).
- The source libraries to use during the build.
- The working catalog to use during the build (optional).

The following diagram shows a simple input file for a source library.

Input File



```
DEFINE_SOURCE_LIBRARIES..
    INTERNAL SOURCE_LIBRARIES = slib

SET_BUILD_CATALOG..
    BUILD_CATALOG = $user.bu

DEFINE_PROCESSOR..
    PROCESSOR = Fortran

DEFINE_PARAMETER_LIST..
    PROCESSOR = Fortran..
    PARAMETER_LIST = 'i = compile b = object_file'

DEFINE_BUILD_TARGET..
    BUILD_TARGET = lib2..
    BUILD_TARGET_KIND = object_library..
    COMPOSITION = 'INCLUDE_DECK DECK = (main..
                        sub1 sub2)'..
    TRANSFORMATION = default
```

Defining a Build Target

To define a build target, use the Build Utility subcommand DEFINE_BUILD_TARGET. At least one build target must be defined in every input file.

A build target is the result of a transformation. For example, an object library can be the build target when you transform a source library. At least one build target must be defined in every Build Utility input file.

To define a build target, use the Build Utility subcommand DEFINE_BUILD_TARGET. The following example defines a build target as an object library named LIB1. LIB1 is dependent on three decks: main, sub1, and sub2. The default transformation is specified.

```
DEFINE_BUILD_TARGET..
    BUILD_TARGET = lib1..
    BUILD_TARGET_KIND = object_library..
    COMPOSITION = 'INCLUDE_DECK DECK = (main ..
                        sub1 sub2)'..
    TRANSFORMATION = default
```

The BUILD_TARGET parameter specifies the name of the build target being defined.

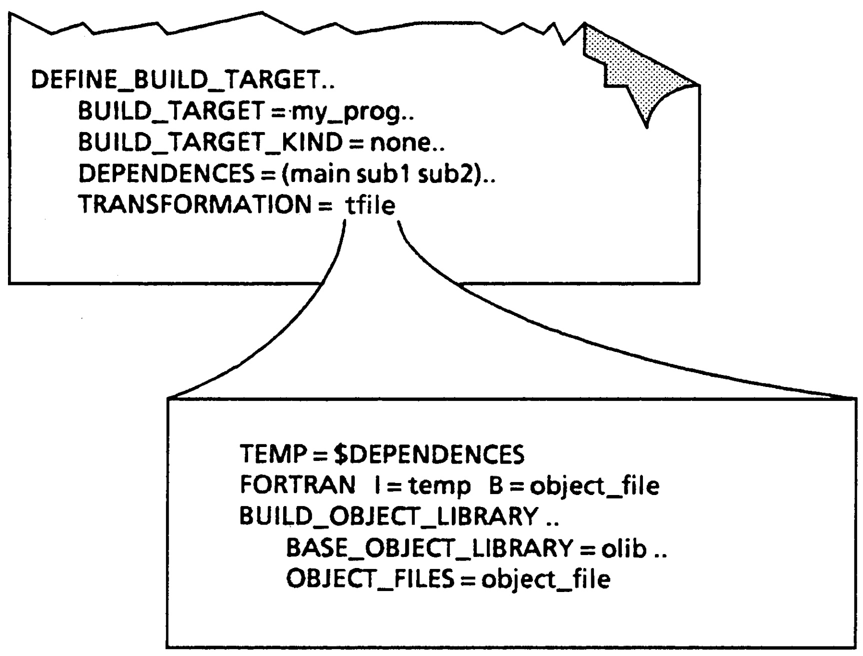
The BUILD_TARGET_KIND parameter tells the Build Utility the type of build target being defined. OBJECT_LIBRARY is a predefined type of build target. If you want to define a build target of a type other than OBJECT_LIBRARY, you may use an appropriate name, or the keyword NONE.

The COMPOSITION parameter is required when the BUILD_TARGET_KIND parameter is assigned OBJECT_LIBRARY. This parameter specifies the selection criteria for the decks that compose the build target.

The final parameter in this example is the TRANSFORMATION parameter. This tells the Build Utility what transformation is required to create the build target. This example specifies the keyword DEFAULT because the Build Utility knows how to handle the transformation of an object library.

If you want to define a build target based on a file system rather than a library, use the DEPENDENCES parameter to list the files that compose the build target. The following example defines a build target, named MY_PROG. MY_PROG is dependent on three files: main, sub1, and sub2.

Input File



```
DEFINE_BUILD_TARGET..
    BUILD_TARGET = my_prog..
    BUILD_TARGET_KIND = none..
    DEPENDENCES = (main sub1 sub2)..
    TRANSFORMATION = tfile
```

```
TEMP = $DEPENDENCES
FORTRAN  I = temp  B = object_file
BUILD_OBJECT_LIBRARY ..
    BASE_OBJECT_LIBRARY = olib ..
    OBJECT_FILES = object_file
```

Transformation File (tfile)

Because MY_PROG is dependent on files rather than decks in a source library, BUILD_TARGET_KIND cannot be specified as OBJECT_LIBRARY. Instead, it is specified as NONE. There is no default transformation for a build target of kind NONE. Therefore, a transformation file is specified. This file contains the transformation required to build the build target.

For a complete description of all parameters for the DEFINE_BUILD_TARGET command, see chapter 5, Commands and Functions.

Defining a Processor

To define a processor, use the Build Utility subcommand DEFINE_PROCESSOR. At least one processor must be defined in the input file, unless a transformation file is specified.

A processor is a command that the Build Utility uses during a transformation. At least one processor must be defined in the input file unless a transformation file is specified. The input file may have several processor definitions. To define a processor, use the Build Utility subcommand DEFINE_PROCESSOR. The following is an example of a processor definition for the FORTRAN compiler.

```
DEFINE_PROCESSOR..
    PROCESSOR = fortran..
    PREPROCESSOR = none..
    DEFAULT_PARAMETER_LIST = default
```

The PROCESSOR parameter specifies the name of the processor being defined.

The PREPROCESSOR parameter specifies the name of a preprocessor to use prior to executing the processor.

The DEFAULT_PARAMETER_LIST parameter specifies the name of the parameter list to use for the processor.

For a complete description of all parameters for the DEFINE_PROCESSOR command, see chapter 5, Commands and Functions.

Using the Predefined Processors

Since most users will be using source and object libraries, the Build Utility has two predefined processors: EXPAND_SOURCE and BUILD_OBJECT_LIBRARY. These processors are used to expand decks into a file and to build an object library from object modules. The Build Utility uses default parameter values for these processors which can be changed using the DEFINE_PARAMETER_LIST command in a Build Utility input file. From PPE, the values can be changed using the EDIT_PARAMETER_LIST command or function key. Neither of these processors needs to be included in your input file unless you want to change the default parameter values.

For a complete description of all the parameters for the predefined processors, see chapter 5, Commands and Functions.

NOTE

The BUILD_OBJECT_LIBRARY processor repeatedly tries to create cycle one of the object library until it is successful. If cycle one of your object library already exists, the build will never terminate. To solve this, either delete your cycle one or copy it to a higher cycle and then delete it.

Defining a Parameter List

Most processors require one or more parameters. Use the DEFINE_PARAMETER_LIST command to assign a particular list of parameters to a processor.

One of the optional components of the Build Utility input file is the DEFINE_PARAMETER_LIST command. This command allows you to assign parameters to a particular processor. These parameters are passed to the processor at execution time. You can define several parameter lists for a single processor. Each parameter list is assigned a name which is referenced in the DEFINE_PROCESSOR command. The following example defines a parameter list for the FORTRAN compiler.

```
DEFINE_PARAMETER_LIST..
    PARAMETER_LIST_NAME = default..
    PARAMETER_LIST = 'i=compile..
                      b=object_file'..
    PROCESSOR = fortran
```

The PARAMETER_LIST_NAME parameter specifies the name of the parameter list being defined. The above example uses the name DEFAULT. This name should match the name that you specify in the DEFAULT_PARAMETER_LIST parameter on the DEFINE_PROCESSOR command.

PARAMETER_LIST specifies a string containing the parameters to pass to the specified processor.

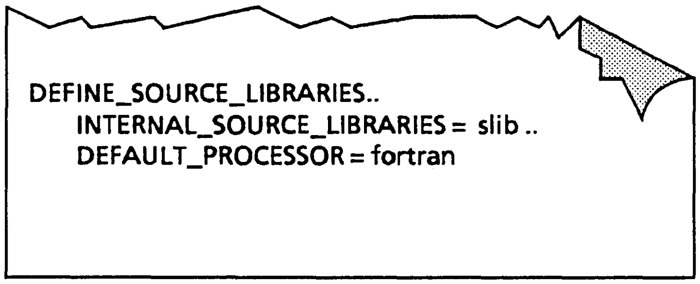
The PROCESSOR parameter specifies the name of the processor to associate with the parameter list.

For a complete description of all parameters for the DEFINE_PARAMETER_LIST command, see chapter 5, Commands and Functions.

Defining the Source Libraries

To define the source libraries to use during the build, use the DEFINE_SOURCE_LIBRARIES command.

The DEFINE_SOURCE_LIBRARIES command allows you to specify the internal and external source libraries to use during the execution of the Build Utility. This command is a required component of the input file, if the build target kind is specified as an object library. The following example defines a source library, called SLIB.



```
DEFINE_SOURCE_LIBRARIES..
    INTERNAL_SOURCE_LIBRARIES = slib ..
    DEFAULT_PROCESSOR = fortran
```

The INTERNAL_SOURCE_LIBRARIES parameter specifies the names of the source libraries to use during the build.

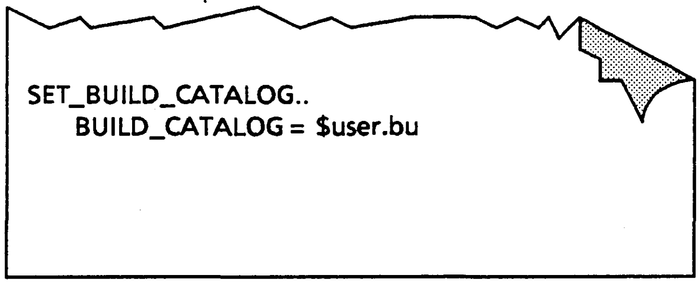
The DEFAULT_PROCESSOR parameter specifies the name of the processor to use during the build when the processor attribute of a deck header is undefined. The above example uses FORTRAN as the default processor.

For a complete description of all parameters for the DEFINE_SOURCE_LIBRARIES command, see chapter 5, Commands and Functions.

Setting a Build Catalog

To specify a catalog to use for the build, use the SET_BUILD_CATALOG command.

The SET_BUILD_CATALOG command specifies the catalog to use during the build. This is an optional component of the Build Utility input file. If you do not specify the catalog to use, the current working catalog is used. The following example sets the build catalog to $USER.BU.



```
SET_BUILD_CATALOG..
    BUILD_CATALOG = $user.bu
```

Getting Online Help

It is possible to get online help for the Build Utility subcommands and functions by starting the Build Utility in terminal input mode.

The Build Utility normally operates in file input mode. All of the inputs to the Build Utility are provided in the input file. However, if you need an online description of the Build Utility subcommands or functions, start the Build Utility in terminal input mode. This method allows you to get command descriptions for all of the Build Utility subcommands. Most Build Utility functions are only available during the transformation of a build target. Therefore, this method provides descriptions for only a few functions. The following example shows how to obtain online help for the Build Utility:

```
/build_software  i=input
BU/display_command_information  define_source_libraries

Names:   define_source_libraries,  defsl

Parameters:

internal_source_libraries, isl: list of file = $required
external_source_libraries, esl: list of file = $optional
analyze_external_source, aes: boolean = false
default_processor, dp: name = $optional
status: (VAR, BY_NAME) status = $optional
BU/quit
--ERROR-- No work can be done by the utility because the INPUT file
did not contain a subcommand that defined a build target.
/
```

The above error occurs every time you quit the Build Utility without defining a build target. You will not want to run the Build Utility in terminal input mode except to get online help because you will have to interactively create an input file every time you run the utility.

The Build Utility and PPE 3

The Build Utility and PPE 3

Several aspects of the Build Utility are specific to PPE. PPE automatically maintains information about the processors, parameter lists, source libraries, and the build catalog for a given library system. PPE also sets up a default build target definition. This chapter describes these PPE defaults. It also provides a description of the PPE screens that affect the way the Build Utility runs.

Declaring an Input File

If you are not satisfied with the PPE default build target, you can create your own input file. This file defines all the build targets for your system.

You can redefine the data dependency of your system by creating an input file. This is a powerful capability. It allows you to define multiple build targets, incorporate complex data dependencies, and create your own transformations. However, if you declare an input file you must provide a full description of all build targets for your system. PPE defines source libraries, processors, parameter lists, and the build working catalog regardless of whether you specify an input file, but defines a build target only when an input file is not specified.

If you want to create your own input file, specify the file name on the Tailor Options screen. You can use the EDIT_FILE utility to create the input file. If you use the EDIT_DECK command, you must remember to extract the deck to a file before starting the Build Utility. The Build Utility does not recognize a deck as an input file.

PPE Screens

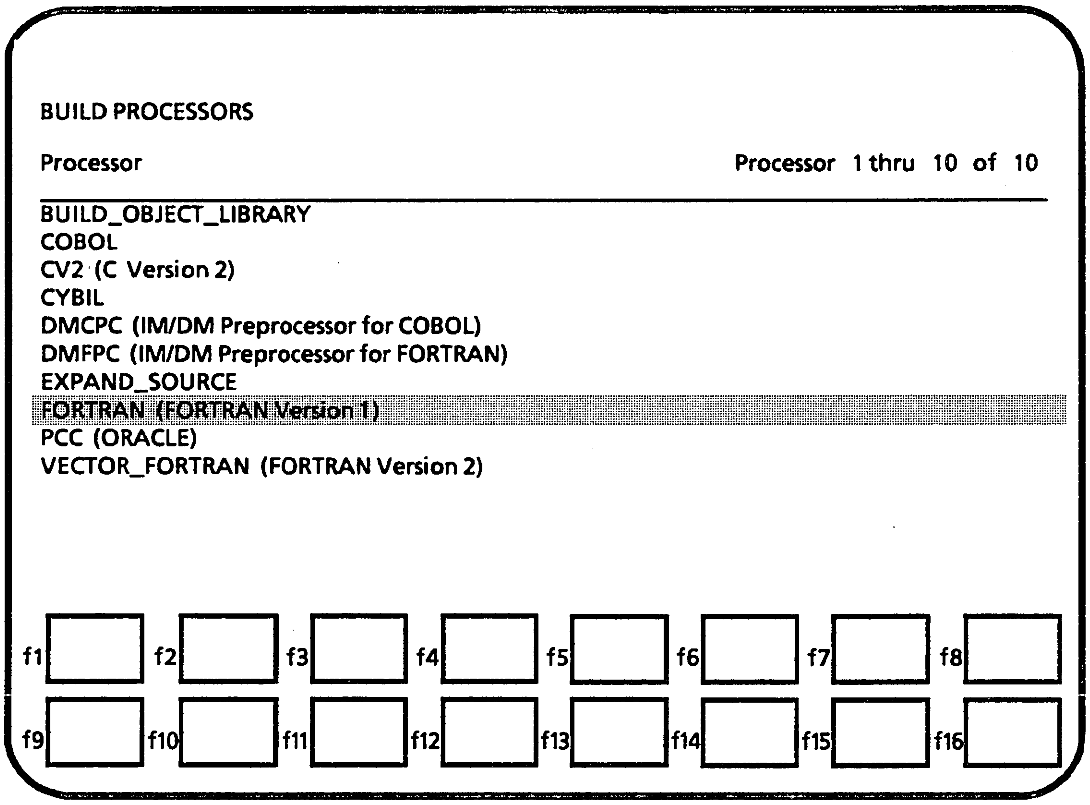
There are four PPE screens that you can use to affect the way the Build Utility runs: Build Processors, Processor Definition, Parameter List Library, and Tailor Options.

This section provides a brief description of the four PPE screens that affect the way the Build Utility runs. These four screens are: Build Processors, Processor Definition, Parameter List Library, and Tailor Options. However, this section does not provide specific information on using or manipulating each screen. This information is provided in the Professional Programming Environment Usage manual.

Build Processors Screen

The Build Processors screen contains a list of all the available processor and preprocessors. The default processor for the current PPE level is highlighted. It is possible to add and delete processors from this list.

The following example shows the Build Processors screen containing the seven predefined processors and three predefined preprocessors.

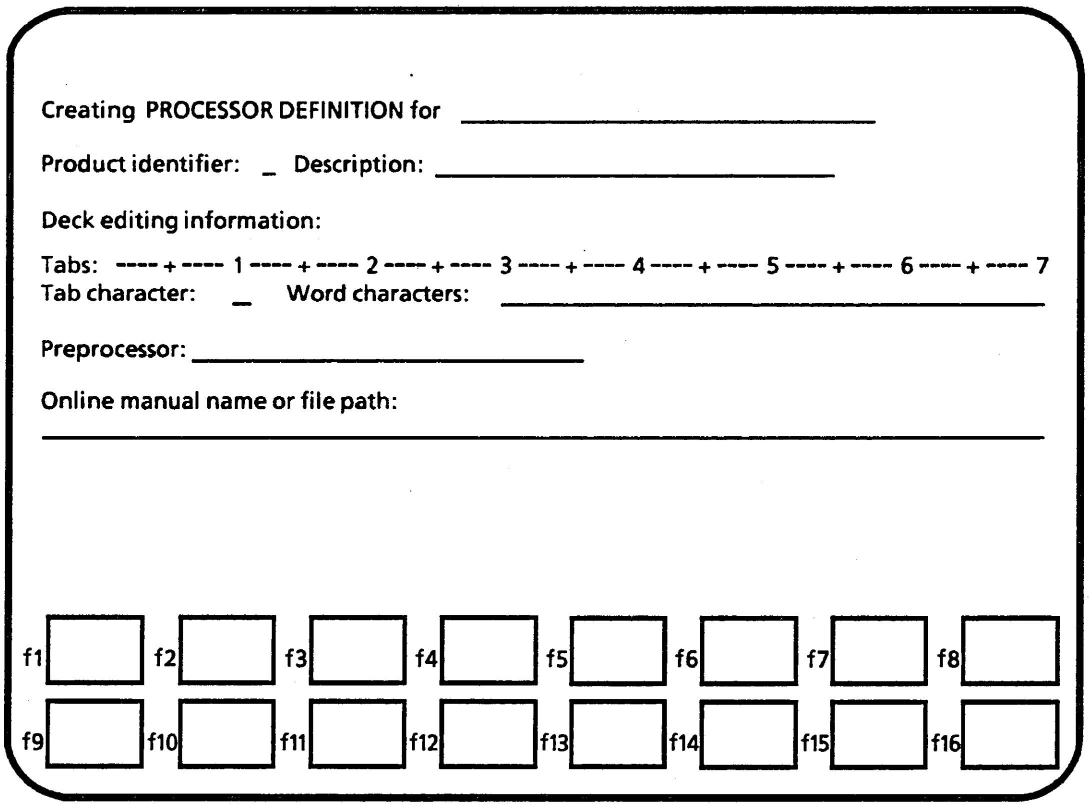


```
BUILD PROCESSORS

Processor                                        Processor 1 thru 10 of 10
--------------------------------------------------------------------------
BUILD_OBJECT_LIBRARY
COBOL
CV2 (C Version 2)
CYBIL
DMCPC (IM/DM Preprocessor for COBOL)
DMFPC (IM/DM Preprocessor for FORTRAN)
EXPAND_SOURCE
FORTRAN (FORTRAN Version 1)
PCC (ORACLE)
VECTOR_FORTRAN (FORTRAN Version 2)
```

Processor Definition Screen

The Processor Definition screen allows you to create a new processor for use by the Build Utility. You can also edit the definitions of existing processors.

The following example shows the Processor Definition screen.

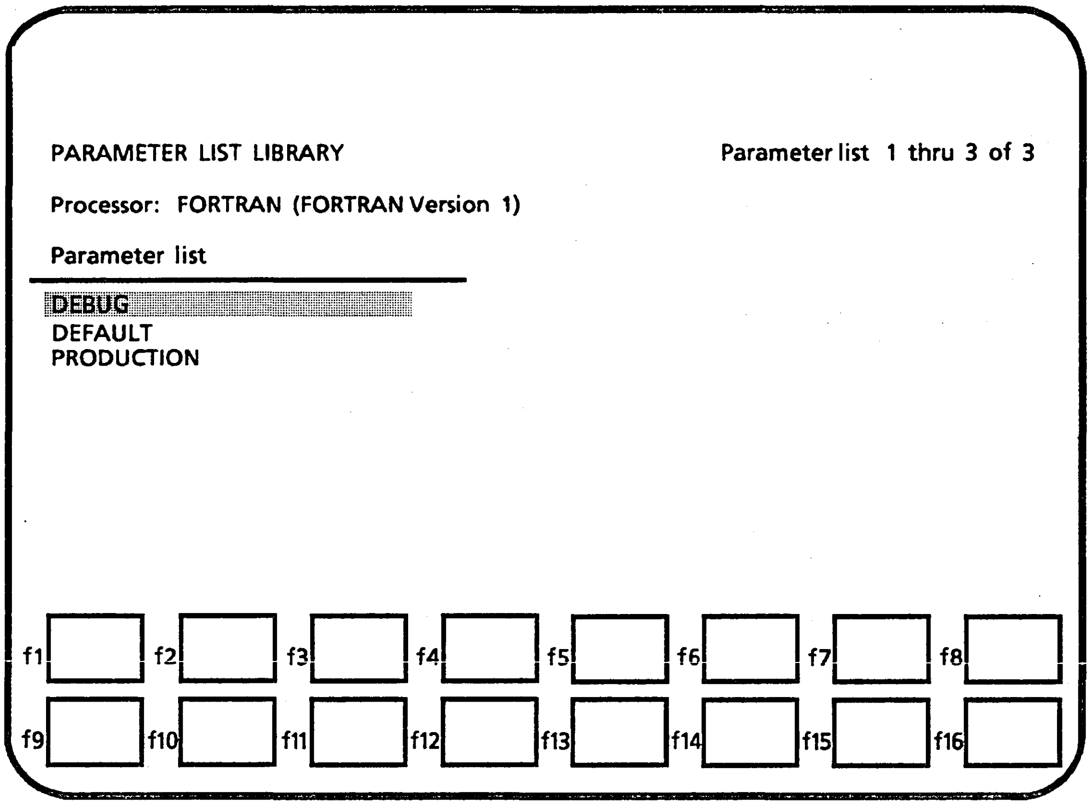


```
Creating PROCESSOR DEFINITION for ______________________________

Product identifier: _  Description: ______________________________

Deck editing information:

Tabs: ----+---- 1 ----+---- 2 ----+---- 3 ----+---- 4 ----+---- 5 ----+---- 6 ----+---- 7
Tab character:  _   Word characters: ________________________________________

Preprocessor: ____________________________

Online manual name or file path:
______________________________________________________________________________
```

Parameter List Library Screen

The Parameter List Library screen lists all the defined parameter lists for a specified processor. From this screen you can create, modify, and delete parameter lists. When creating or modifying a parameter list in PPE, you are presented with the full NOS/VE parameter list for the specified processor.

The following example shows the default Parameter List Library for the FORTRAN processor.



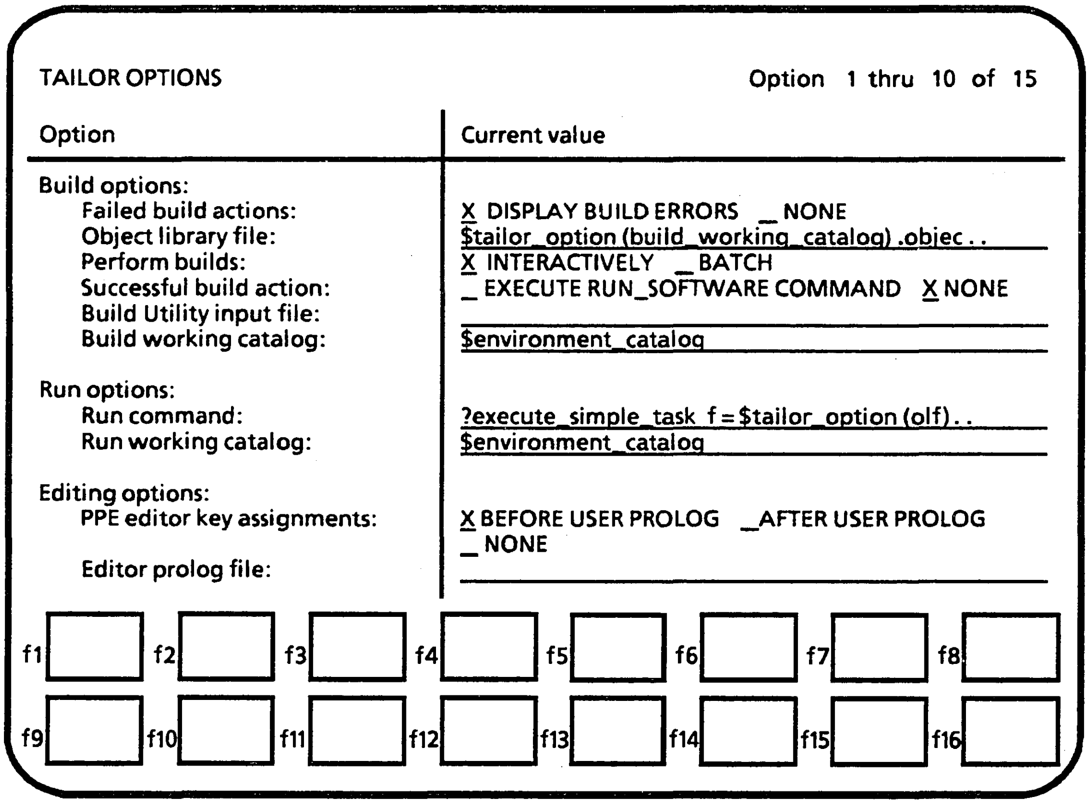
```
PARAMETER LIST LIBRARY                              Parameter list 1 thru 3 of 3

Processor: FORTRAN (FORTRAN Version 1)

Parameter list
DEBUG
DEFAULT
PRODUCTION
```

Tailor Options Screen

The Tailor Options screen allows you to change several parameters that affect the way the Build Utility runs. This screen allows you to specify the following:

- The name of the object library file.
- The mode in which builds are performed (interactive or batch).
- The build working catalog.
- A Build Utility input file.

The following example shows the Tailor Options screen with the default values.



```
TAILOR OPTIONS                                     Option  1 thru  10  of  15

Option                              | Current value
--------------------------------------------------------------------------------
Build options:                      |
   Failed build actions:            | X DISPLAY BUILD ERRORS  _ NONE
   Object library file:             | $tailor_option (build_working_catalog) .objec . .
   Perform builds:                  | X INTERACTIVELY  _ BATCH
   Successful build action:         | _ EXECUTE RUN_SOFTWARE COMMAND  X NONE
   Build Utility input file:        | ______________________________________
   Build working catalog:           | $environment_catalog

Run options:                        |
   Run command:                     | ?execute_simple_task f=$tailor_option (olf) . .
   Run working catalog:             | $environment_catalog

Editing options:                    |
   PPE editor key assignments:      | X BEFORE USER PROLOG   _AFTER USER PROLOG
                                    | _ NONE
   Editor prolog file:              | ______________________________________
```

Getting Online Help

It is possible to get an online description of the Build Utility subcommands and functions by starting the Build Utility in terminal input mode.

The Build Utility normally operates in file input mode. All of the input to the Build Utility is provided by PPE and the input file. However, if you need an online description of the Build Utility subcommands or functions, start the utility in terminal input mode. This method provides descriptions of all the Build Utility subcommands. However, most Build Utility functions are only available during the execution of a transformation. Therefore, this method provides descriptions for only a few Build Utility functions. To do this, use the Tailor Options screen to define the input file as INPUT. Once the input file is set to input, start the Build Utility by using one of the build commands or function keys. The following example shows how to get online help for the Build Utility subcommands and functions:

```
INCF/display_command_information  define_source_libraries

Names:   define_source_libraries,  defsl

Parameters:

internal_source_libraries, isl: list of file = $required
external_source_libraries, esl: list of file = $optional
analyze_external_source, aes: boolean = false
default_processor, dp: name = $optional
status: (VAR, BY_NAME) status = $optional
INCF/quit
--ERROR-- No work can be done by the utility because the INPUT file
did not contain a subcommand that defined a build target.
/
```

The above message occurs every time you quit the Build Utility without defining a build target. You will not want to run the Build Utility in terminal input mode except to get online help because you will have to interactively create an input file every time you run the utility. An optional method for getting online help in PPE is to enter the BUILD_SOFTWARE command with the INPUT parameter specified as input from the home line.

NOTE

This section does not refer to the INTERACTIVE and BATCH fields on the Tailor Options screen. These fields determine whether to run the Build Utility interactively, but do not affect the mode of input. However, to get online help you must run the Build Utility interactively.

Advanced Usage 4

Advanced Usage 4

This chapter describes some of the more advanced features and applications of the Build Utility. These features include preprocessors, using the PROCESSOR attribute, multiple processors, multiple build targets, layering, and complex transformations.

Defining a Preprocessor

One of the optional parameters of the DEFINE_PROCESSOR command allows you to define a preprocessor.

A preprocessor is the name of a command which prepares source text for input to a specified processor. The preprocessor parameter is needed when the source text must be prepared in some way before being passed to the main processor. An example of a preprocessor is the IM/DM command DMCPC. In this case, the precompiler scans the source module for DM commands that are embedded in the COBOL code. This DM code must be converted to COBOL code before it is sent to the COBOL compiler.

The following is an example of a processor definition which includes a preprocessor.

```
DEFINE_PROCESSOR..
    PROCESSOR = cobol..
    PREPROCESSOR = DMCPC..
    DEFAULT_PARAMETER_LIST = default
```

Using the PROCESSOR Attribute

The Build Utility allows you to use the PROCESSOR attribute of a deck header to specify both a processor and a parameter list to use for the build.

Every deck header has a PROCESSOR attribute. This attribute accepts a string value. This string usually specifies the name of the processor to use for building that deck. However, the Build Utility allows you to use this attribute to specify both the name of a processor and the name of a parameter list.

To do this, the Build Utility checks the PROCESSOR attribute to see if it satisfies the following format.

```
processor,p : name=$optional
parameter_list_name, pln : name=$optional
```

If the Build Utility determines that the value satisfies this format, it uses the processor and the parameter list that are specified. If either one or both are not specified, the Build Utility uses the default processor or the default parameter list or both to build the deck. The following are examples of values that satisfy the above format:

```
fortran default
p=cobol pln=list2
pln=plist1
```

The Build Utility executes the following INCLUDE_COMMAND on the values that satisfy the Build Utility format.

```
include_command 'processor '//$parameter_list_value (p,pln)
```

In this example, `processor` refers to the name of the processor specified. If the value of the PROCESSOR attribute does not satisfy the above format, the Build Utility assumes that the value contains the complete command for transforming that deck and executes an INCLUDE_COMMAND on the value. For example, the following value does not satisfy the Build Utility format.

```
fortran i=compile b=object_file
```

Therefore, the Build Utility executes the INCLUDE_COMMAND on the entire value.

NOTE

When using PPE, you get a warning message if the value you specify in the PROCESSOR attribute does not appear on the Build Processors screen. However, the Build Utility still processes the value as described above.

Defining Multiple Processors

The Build Utility allows you to define multiple processors in a single input file.

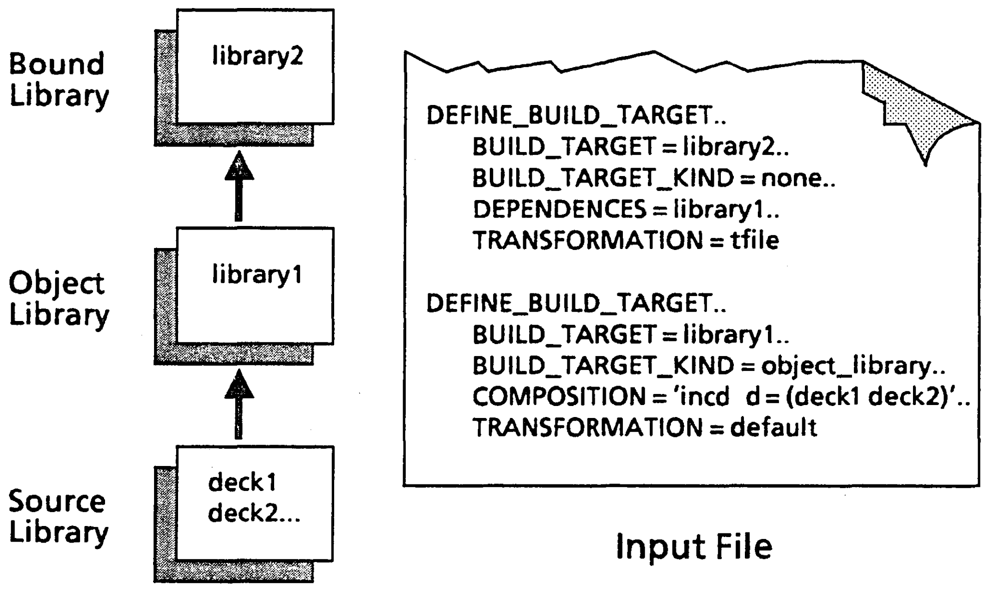
The Build Utility uses the PROCESSOR attribute on the individual deck header to determine which processor to use for the transformation. Therefore, you can combine object text from different processors into one object library by specifying different processor names on the deck header. For example, code from a FORTRAN, COBOL, and CYBIL compiler can be combined into one object library using the Build Utility.

In a file system, using multiple processors for a given build target is more difficult. The Build Utility cannot determine which processor to use for a given file simply by looking at the file. To handle multiple processors in a file system, you can define a separate build target for each processor. Then define another build target that is composed of the individual object modules.

Defining Multiple Build Targets

The Build Utility allows you to define multiple build targets.

You are allowed to have multiple occurrences of the DEFINE_BUILD_TARGET command in a Build Utility input file. For example, your software system could consist of two libraries: an unbound library and a bound version. The following diagram shows a system with more than one build target. This example shows two build targets: an object library and a bound library.



```
DEFINE_BUILD_TARGET..
    BUILD_TARGET = library2..
    BUILD_TARGET_KIND = none..
    DEPENDENCES = library1..
    TRANSFORMATION = tfile

DEFINE_BUILD_TARGET..
    BUILD_TARGET = library1..
    BUILD_TARGET_KIND = object_library..
    COMPOSITION = 'incd  d = (deck1 deck2)'..
    TRANSFORMATION = default
```

The order in which the build targets are defined in the input file is important. By default, the Build Utility analyzes only the first build target in the input file. However, the process of analyzing a build target involves first analyzing all the decks or files on which it is dependent. Therefore, by defining library2 first, the Build Utility analyzes library1. However, if you reverse the definitions, only library1 is analyzed.

You can also use the BUILD_TARGET parameter on one of the build commands (BUILD_SOFTWARE, BUILD_CHANGED_DECKS, or BUILD_DECKS) to specify which build targets to build. Specifying ALL causes the Build Utility to build all targets, regardless of order.

Defining a Layered System

The Build Utility allows you to declare and maintain a layered system of build targets.

In a development environment, object libraries are often maintained at several levels. For example, one object library exists at the programmer level, another at the coordinator level, and a final version exists at integration. This concept is called layering. The Build Utility handles layering by allowing you to declare a list of object libraries associated with each build target. This is done using the LAYERS parameter of the Build Utility subcommand DEFINE_BUILD_TARGET.

When the Build Utility encounters a layered system, it searches each level to find the first occurrence of a given module. The search begins with the current build target and then continues with each object library in the order it is declared in the LAYERS parameter. When the Build Utility finds the first occurrence of a module, it uses it as the basis for analysis. If the module is younger than the next occurrence, the Build Utility performs the transformation. If it is not younger, no transformation is performed.

Creating Complex Transformations

The Build Utility allows you to create complex transformations. These transformations are specified in the transformation file.

The TRANSFORMATION parameter of the Build Utility subcommand DEFINE_BUILD_TARGET accepts a file name. This file, called the transformation file, can contain NOS/VE commands and the Build Utility commands and functions. Since you can write NOS/VE procedures that perform almost any task, you can create almost any transformation. The Build Utility places no limit on the size or complexity of the transformation file. You are limited only by your ability to define a particular transformation.

Commands and Functions 5

Commands and Functions 5

Command and Subcommands

This section contains descriptions of the Build Utility command and subcommands. The commands are presented in alphabetical order. Each description provides the following information:

- Purpose of the command.
- Format of the command.
- Descriptions of the command parameters.

Optionally, the command descriptions may contain remarks and examples of command usage.

The command format provides the full command name, any abbreviations of the command, and the parameters for the command.

The STATUS parameter on the Build Utility commands is the same STATUS parameter available on all NOS/VE commands. If the STATUS parameter is specified, it must be specified by name, not by position.

Using Build Utility Subcommands

The Build Utility subcommands are used in the Build Utility input file. The only command that can be entered directly from NOS/VE is the Build Utility command BUILD_SOFTWARE.

BUILD _SOFTWARE
Build Utility Command

Purpose Initiates the Build Utility.

Format

BUILD _SOFTWARE or
BUIS
  INPUT=file
  BUILD _TARGETS=list of file or keyword
  DECKS=list of name or keyword
  EXECUTE _TRANSFORMATIONS=boolean
  DISPLAY_OPTIONS=list of keyword
  OUTPUT=file
  ERRORS=file
  STATUS=status variable

Parameters

INPUT or I

Specifies a file that describes your file system or library to the Build Utility. This file may contain NOS/VE commands and Build Utility subcommands.

This is a required parameter.

BUILD _TARGETS or *BUILD _TARGET* or *BT*

Specifies which build targets from the input file should be analyzed.

You can reference the build targets by name or use one of the following keywords:

FIRST

Specifies the first build target in the input file.

ALL

Specifies all build targets in the input file.

The default is FIRST.

DECKS or *D*

Specifies the decks to build.

You can specify a deck name, a list of deck names, or the keyword ALL.

ALL

Specifies all decks.

There are only three occasions when you will want to specify this parameter.

- When you want a full build.
- When you know that a deck was changed in such a way that the object code will not be affected. For example, changing a comment in a program does not affect the execution of the object code.
- When you know exactly which decks need to be built.

By default, the Build Utility determines which decks are out of date by comparing the date/time stamp on the source deck with the date/time stamp on the build target.

EXECUTE_TRANSFORMATIONS or *ET*

Specifies whether to execute the transformations for an out-of-date build target.

This parameter accepts a boolean value. If you specify FALSE, the transformations are not made. Only the analysis phase of the build is executed. This allows you to determine which decks would be built without actually performing a build.

By specifying the DISPLAY_OPTIONS parameter, you can cause the Build Utility to display the out-of-date build targets and the reasons they were found to be out of date.

The default value is TRUE.

DISPLAY_OPTIONS or *DO*

Specifies the information to display about the build. The Build Utility writes this information to the file specified by the OUTPUT parameter.

Specify one of the following keywords:

ANALYSIS_TRACE or AT

Writes the steps taken by the Build Utility during the build.

ANALYSIS_RESULTS or AR

Writes the results of the analysis phase of the build.

NONE

Indicates that no information is written.

The default value is NONE.

OUTPUT or *O*

Specifies the name of the file to which the information generated by the DISPLAY_OPTIONS parameter is written.

The default value is $OUTPUT.

ERRORS or *E*

Specifies the name of the file to which error messages are written.

The default value is $ERRORS.

Remarks This is the only Build Utility command that can be entered directly from the command prompt.

Example The following example uses the BUILD_SOFTWARE command to start the Build Utility. IFILE is the name of the Build Utility input file.

```
/build_software input=ifile display_options=analysis_trace
```

or abbreviated,

```
/buis ifile do=at
```

DEFINE_BUILD_TARGET Build Utility Subcommand

Purpose Defines a build target by specifying the files it depends on and the transformation to be performed when the target is found to be out of date.

Format

DEFINE_BUILD_TARGET or
DEFBT
  BUILD_TARGET=file
  BUILD_TARGET_KIND=name or keyword
  DEPENDENCES=list of file
  COMPOSITION=keyword or file or string
  COMPOSITION_MAP=file
  LAYERS=list of file
  TRANSFORMATION=keyword or file or string
  STATUS=status variable

Parameters

BUILD_TARGET or **BT**

Specifies the build target name. A file name or library must be specified.

This is a required parameter.

BUILD_TARGET_KIND or *BTK*

Specifies the type of the build target. Specify an appropriate name or one of the following keywords:

OBJECT_LIBRARY or OL

Indicates that the build target type is an object library.

NONE

No type is assigned to the build target.

The default is NONE.

DEPENDENCES or *D*

Specify a list of files that the build target depends upon. Files that are specified in this parameter can also be build targets.

If omitted, the Build Utility assumes that the build target is dependent upon decks rather than files and uses the COMPOSITION parameter to determine the decks.

COMPOSITION or *C*

Specifies the expandable decks that compose the build target. Specify a string or a file containing SCU selection criteria commands, or the following keyword:

MAPPED_DECKS_ONLY or MDO

Indicates that the build target is only dependent on the decks specified by the COMPOSITION_MAP parameter.

If omitted, the Build Utility assumes that the build target is dependent on files rather than decks and uses the DEPENDENCES parameter to determine the files.

COMPOSITION_MAP or *CM*

Specifies a file containing a list of source decks mapped to object library entries. Each mapping has the following format:

```
deck:name = $required  object_library_entry:name = $optional
```

By default, the name in the object library matches the name of the deck.

LAYERS or *L*

Specifies a list of files that comprise the layers of a system. These layers are searched in order, starting with the build target itself, to find the first occurrence of a module. The Build Utility uses this module as the basis for its analysis.

By default, no layers are defined.

TRANSFORMATION or **T**

Specifies the transformation to perform when the build target is out of date. Specify a string or file which contains one or more NOS/VE commands, or the following keyword:

This is a required parameter.

DEFAULT or D

Specifies that transformation is determined by the BUILD_TARGET_KIND parameter. In order to use DEFAULT for the TRANSFORMATION parameter, the BUILD_TARGET_KIND must not be NONE.

Remarks

- This command can only be used in a Build Utility input file.
- A build target can be any file, including an object library.
- A build target can be dependent on other build targets.
- You must specify either the DEPENDENCES parameter or the COMPOSITION parameter on this command.

Example

The following example defines a build target named TARGET1 as an object library.

```
define_build_target..
  build_target = target1..
  build_target_kind = object_library..
  composition = 'incd d = (deck1 deck2 deck3)'..
  transformation = default
```

or abbreviated,

```
defbt..
  bt = target1..
  btk = ol..
  c = 'incd d = (deck1 deck2 deck3)'..
  t = default
```

DEFINE_PARAMETER_LIST
Build Utility Subcommand

Purpose Defines the parameters to pass to the specified processor during a transformation.

Format **DEFINE_PARAMETER_LIST** or
DEFPL
PARAMETER_LIST_NAME=name
PARAMETER_LIST=string
PROCESSOR=name
STATUS=status variable

Parameters *PARAMETER_LIST_NAME* or *PLN*

Specifies a name to associate with the parameter list. The name must be unique to the specified processor.

The default is DEFAULT.

PARAMETER_LIST or **PL**

Specifies a string containing the parameters to pass to a given processor.

This is a required parameter.

PROCESSOR or **P**

Specifies a processor to associate with the parameter list. The processor specified by this parameter must be defined with a DEFINE_PROCESSOR command *prior* to being referenced by this parameter.

This is a required parameter.

Remarks This command can only be used in a Build Utility input file.

Example The following example defines a parameter list to pass to the COBOL processor:

```
define_parameter_list..
   parameter_list_name=plist1..
   parameter_list='i=compile bo=object_file'..
   processor=cobol
```

or abbreviated,

```
defpl..
   pln=plist1..
   pl='i=compile bo=object_file'..
   p=cobol
```

DEFINE_PROCESSOR
Build Utility Subcommand

Purpose Defines a processor to use during the execution of a transformation.

Format **DEFINE_PROCESSOR** or
DEFP
  PROCESSOR=name
  PREPROCESSOR=name or keyword
  DEFAULT_PARAMETER_LIST=name
  STATUS=status variable

Parameters **PROCESSOR** or **P**

Specifies the name of the processor to define.

This is a required parameter.

PREPROCESSOR or *PP*

Specifies a preprocessor to use prior to executing the processor. A preprocessor prepares the source text for the main processor. An example of a preprocessor is DMFPC, which converts all the embedded DM commands in source code to FORTRAN code.

The default is NONE.

DEFAULT_PARAMETER_LIST or *DPL*

Specifies the name of the parameter list to use by default for the processor. The Build Utility uses this parameter list when the processor attribute of a deck header does not specify a parameter list. The parameter list must be defined using the DEFINE_PARAMETER_LIST command.

The default is DEFAULT.

Remarks

- This command can only be used in a Build Utility input file.
- This command must be specified before specifying the DEFINE_PARAMETER_LIST command.

Example The following example defines a COBOL processor and uses the default parameter list PLIST1:

```
define_processor..
   processor=cobol..
   default_parameter_list=plist1
```

or abbreviated,

```
defp..
   p=cobol..
   dpl=plist1
```

DEFINE_SOURCE_LIBRARIES Build Utility Subcommand

Purpose Specifies the internal and external source libraries to use during the build.

Format **DEFINE_SOURCE_LIBRARIES** or
DEFSL
  INTERNAL_SOURCE_LIBRARIES=list of file
  EXTERNAL_SOURCE_LIBRARIES=list of file
  ANALYZE_EXTERNAL_SOURCE=boolean
  DEFAULT_PROCESSOR=name
  STATUS=status variable

Parameters **INTERNAL_SOURCE_LIBRARIES** or **ISL**

Specifies one or more source libraries to use during the build. The Build Utility searches these libraries in the order they are specified.

This is a required parameter.

EXTERNAL_SOURCE_LIBRARIES or *ESL*

Specifies one or more source libraries containing decks that are external to the system being built, but are referenced by internal decks. The Build Utility searches these libraries in the order they are specified.

By default, no external source libraries are searched.

ANALYZE_EXTERNAL_SOURCE or *AES*

Specifies whether to include the external decks in the dependency analysis.

The default is FALSE.

DEFAULT_PROCESSOR or *DP*

Specifies the processor to use during a build when the processor attribute in a deck header is undefined. The processor must be defined using the DEFINE_PROCESSOR command *prior* to being referenced by this parameter.

By default, the processor is defined by each deck's PROCESSOR attribute.

Remarks

- This command can only be used in a Build Utility input file.
- This command is a required component of the input file when any of the build targets are defined as object libraries.

Example The following example defines a source library called SOURCE_LIB and specifies COBOL as the default processor:

```
define_source_libraries..
   internal_source_libraries=source_lib..
   default_processor=cobol
```

or abbreviated,

```
defsl..
   isl=source_lib..
   dp=cobol
```

SET_BUILD_CATALOG
Build Utility Subcommand

Purpose Specifies the catalog to use during the build.

Format **SET_BUILD_CATALOG** or
SETBC
    BUILD_CATALOG=catalog name
    STATUS=status variable

Parameters *BUILD_CATALOG* or *BC*

Specifies the full path name of the catalog to use during the build.

The default is the default working catalog.

Remarks This command can only be used in a Build Utility input file.

This page intentionally left blank.

Functions

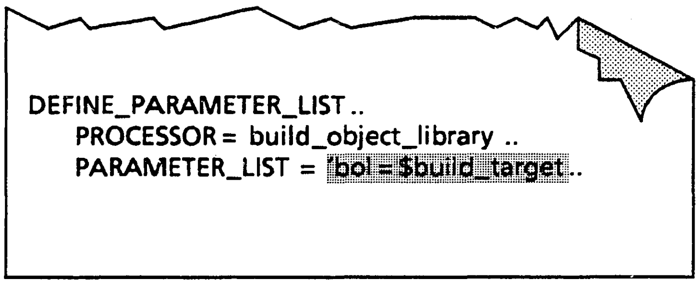
This section contains descriptions of the Build Utility functions. These descriptions are presented alphabetically. Each description provides the following information:

- Purpose of the function.
- Format of the function.
- Descriptions of the function parameters (if any).

These descriptions may contain remarks and examples of the function's usage.

Using Build Utility Functions

Functions are used in the Build Utility input file or a transformation file. The values that some return depend on the build target whose transformation is executing. The following example shows how to use a function in the Build Utility input file:



```
DEFINE_PARAMETER_LIST..
    PROCESSOR= build_object_library ..
    PARAMETER_LIST = 'bol=$build_target..
```

Build Utility functions are only available during the execution of the Build Utility. However, some commands start new tasks (FORTRAN, for example). Build Utility functions are not available while these new tasks are executing, nor can they be passed as parameters to the new task. For example, you cannot include the following statement in an input file.

```
fortran ~~i=$dependences~~ b=object_file
```

Instead, you need to call the function first, then call the FORTRAN compiler.

```
temp=$dependences
fortran i=temp b=object_file
```

$ALTERNATE_SOURCE_LIBRARIES Build Utility Function

Purpose Returns a list of the internal and external source libraries excluding the first internal source library given.

Format **$ALTERNATE_SOURCE_LIBRARIES** or **$ASL**

Remarks If no alternate source libraries are specified, an empty list is returned.

Example

```
define_source_libraries..
  internal_source_libraries=(lib1,lib2)

display_value  $alternate_source_libraries
```

Result

```
:V01.kevin.lib2
```

$BASE_SOURCE_LIBRARY
Build Utility Function

Purpose Returns the name of the first internal source library defined in the DEFINE_SOURCE_LIBRARIES command.

Format **$BASE_SOURCE_LIBRARY** or
$BSL

Example

```
define_source_libraries..
  internal_source_libraries=(lib1,lib2)

display_value  $base_source_library
```

Result

```
:V01.kevin.lib1
```

$BUILD_CATALOG
Build Utility Function

Purpose Returns the name of the catalog used during the build.

Format **$BUILD_CATALOG** or
$BC

Remarks If the input file does not specify the SET_BUILD_CATALOG command, this function returns the name of the default working catalog.

Example

```
set_working_catalog..
  working_catalog=$user.exmps

display_value  $build_catalog
```

Result

```
:V01.kevin.exmps
```

$BUILD_TARGET
Build Utility Function

Purpose Returns the name of the build target whose transformation is currently executing.

Format **$BUILD_TARGET** or **$BT**

Example

```
define_build_target..
  build_target=target1..
  build_target_kind=none..
     ⋮

display_value  $build_target
```

Result

```
:V01.kevin.target1
```

$BUILD_TARGET_KIND Build Utility Function

Purpose Returns the type of build target whose transformation is currently executing. The value returned is the name specified in the definition of the build target.

Format **$BUILD_TARGET_KIND** or **$BTK**

Example

```
define_build_target..
  build_target=lib1..
  build_target_kind=object_library
     ⋮

display_value  $build_target_kind
```

Result

```
object_library
```

$BUILD_TARGET_LAYERS
Build Utility Function

Purpose Returns the file reference for every layer of the specified build target.

Format

$BUILD_TARGET_LAYERS or
$BTL
  (*BUILD_TARGET=name*)

Parameters *BUILD_TARGET*

Specifies the build target to use.

If the specified file is not a build target, the function returns the file that was given. If no file is specified, the function returns the name of the build target whose transformation is currently executing.

$CHANGED_DECKS Build Utility Function

Purpose

Returns the names of the expandable decks that compose the build target whose transformation is currently executing.

Format

$CHANGED_DECKS or
$CD

Remarks

The value returned depends on the value of the DECKS parameter of the BUILD_SOFTWARE command. If ALL was specified, the function returns a list of all decks that compose the current build target. If no value for the DECKS parameter was specified, the function returns a list of decks from the current build target that are out of date. If a deck name or list of decks was specified in the DECKS parameter, the function returns a list of these decks and any other decks that compose the build target.

Example

```
build_software i=infile d=(deck1 deck2 deck3)

display_value  $changed_decks
```

Result

```
deck1
deck2
deck3
```

$COMPOSITION
Build Utility Function

Purpose Returns a list of all decks that compose the build target whose transformation is currently executing.

Format **$COMPOSITION** or
$C

Remarks If the COMPOSITION parameter on the DEFINE_BUILD_TARGET command was not specified, no value is returned.

Example

```
define_build_target..
  build_target=target1..
  build_target_kind=ol..
  composition='incd d=(deck1 deck2)'..
    :

display_value  $composition
```

Result

```
deck1
deck2
```

$COMPOSITION_MAP Build Utility Function

Purpose Returns a list of the decks and their corresponding object library entries that comprise the build target whose transformation is currently executing.

Format **$COMPOSITION_MAP** or **$CM**

Remarks If the COMPOSITION_MAP parameter on the DEFINE_BUILD_TARGET command was not specified, an empty list is returned.

Example This example uses the following composition map file:

```
   deck1 ent1
   deck2 ent2
   deck3 ent3

display_value $composition_map
```

Result

```
deck1
ent1
deck2
ent2
deck3
ent3
```

$DEPENDENCES Build Utility Function

Purpose Returns the list of files that the build target whose transformation is currently executing depends upon.

Format **$DEPENDENCES** or
$D
(*KIND=keyword*)

Parameters *KIND*

Specifies the files to return.

Specify one of the following keywords:

YOUNGER_THAN_TARGET or YTT

Returns only files that are younger than the target (the changed files).

ALL

Returns all files referenced in the DEPENDENCES parameter.

The default is YOUNGER_THAN_TARGET.

Example In the following example, the build catalog is :V01.kevin.

```
define_build_target..
  build_target=target1..
  build_target_kind=none..
  dependences=(file1 file2)
     :

display_value $dependences(all)
```

Result

```
:V01.kevin.file1
:V01.kevin.file2
```

$DISPLAY_OPTIONS Build Utility Function

Purpose	Returns the display options specified for the build.
Format	**$DISPLAY_OPTIONS** or **$DO**
Remarks	If no display options were specified, an empty list is returned.

$ERRORS_FILE Build Utility Function

Purpose Returns the name of the file containing the error messages from the build.

Format **$ERRORS_FILE** or
$EF

Example The following example assumes that no errors file was specified on the BUILD_SOFTWARE command. Therefore, the default is used.

```
display_value $errors_file
```

Result

```
:$local.$error.1
```

$EXTERNAL_SOURCE_LIBRARIES Build Utility Function

Purpose Returns a list of external source libraries specified for the build.

Format **$EXTERNAL_SOURCE_LIBRARIES** or
$ESL

Remarks If no external source libraries are specified, an empty list is returned.

Example In the following example, the build catalog is :V01.kevin.

```
define_source_libraries..
  internal_source_libraries=slib..
  external_source_libraries=(lib3 lib4)..
    :

display_value $external_source_libraries
```

Result

```
:V01.kevin.lib3
:V01.kevin.lib4
```

$INTERNAL_SOURCE_LIBRARIES Build Utility Function

Purpose Returns a list of internal source libraries specified for the build.

Format **$INTERNAL_SOURCE_LIBRARIES** or **$ISL**

Example In the following example, the build catalog is :V01.kevin.

```
define_source_libraries..
  internal_source_libraries=slib..
  external_source_libraries=(lib3 lib4)..
     ⋮

display_value $internal_source_libraries
```

Result

```
:V01.kevin.slib
```

$LAYERS
Build Utility Function

Purpose Returns a list of files that comprise the layers of the build target whose transformation is currently executing.

Format **$LAYERS** or
$L

Remarks If no layers are specified, an empty list is returned.

Example In the following example, the build catalog is :V01.kevin.

```
define_build_target..
  build_target=target1..
  build_target_kind=object_library..
  layers=(file1 file2)
    ⋮

display_value $layers
```

Result

```
:V01.kevin.file1
:V01.kevin.file2
```

$OUTPUT_FILE
Build Utility Function

Purpose Returns the name of the output file specified for the build.

Format **$OUTPUT_FILE** or
$OF

Remarks If no output file is specified, $OUTPUT is returned.

Example The following example assumes that no output file was specified on the BUILD_SOFTWARE command. Therefore, the default is used.

```
display_value $output_file
```

Result

```
:$local.$ouput.1
```

$PARAMETER_LIST_VALUE Build Utility Function

Purpose Returns a string containing the list of parameters to pass to the processor.

Format **$PARAMETER_LIST_VALUE** or
$PLV
(*PROCESSOR=name or keyword*
PARAMETER_LIST_NAME=name or keyword)

Parameters *PROCESSOR*

Specifies the name of the processor to use. To use the default processor established for this build, specify the keyword DEFAULT_PROCESSOR.

The default is DEFAULT_PROCESSOR.

PARAMETER_LIST_NAME

Specifies the name of the parameter list. To use the default parameter list for the specified processor, specify the keyword DEFAULT_PARAMETER_LIST.

The default is DEFAULT_PROCESSOR.

Example

```
display_value $parameter_list_value(expand_source default)
```

Result

```
d=$changed_decks b=$base_source_library ab=$alternate_source_libraries l=
$output_file e=$errors_file
```

$PROCESSOR_ATTRIBUTE Build Utility Function

Purpose Returns the name of the preprocessor or the default parameter list for the specified processor.

Format **$PROCESSOR_ATTRIBUTE** or
$PA
  (PROCESSOR = name
  ATTRIBUTE = keyword)

Parameters **PROCESSOR**

Specifies the name of the processor.

This is a required parameter.

ATTRIBUTE

Specifies the processor attribute.

Enter one of the following keywords:

PREPROCESSOR or PP

Returns the name of the preprocessor associated with the specified processor. If the processor does not have a preprocessor assigned to it, NONE is returned.

DEFAULT_PARAMETER_LIST or DPL

Returns the name of the default parameter list for the specified processor. If no default parameter list is specified, UNDEFINED is returned.

This is a required parameter.

Example

```
display_value $processor_attribute(expand_source default_parameter_list)
```

Result

```
default
```

$UNKNOWN_LIBRARY_ENTRIES Build Utility Function

Purpose	Returns a list of all object library modules for which no source deck is present in the composition of the build target.
Format	**$UNKNOWN_LIBRARY_ENTRIES** or **$ULE**
Remarks	If no unknown library modules are found, an empty list is returned.

Build Utility Processors

This section contains descriptions of two Build Utility processors. These processors are pre-defined by the Build Utility.

Each description provides the following information:

- Purpose of the command.
- Format of the command.
- Descriptions of the command parameters.

Optionally, the command descriptions may contain remarks and examples of command usage.

The format provides the full processor name, any abbreviations of the processor, and the parameters for the processor.

The STATUS parameter on Build Utility commands is the same STATUS parameter available on all NOS/VE commands. If the STATUS parameter is specified, it must be specified by name, not by position.

BUILD_OBJECT_LIBRARY
Build Utility Processor

Purpose Builds an object library.

Format **BUILD_OBJECT_LIBRARY** or
BUIOL
  BASE_OBJECT_LIBRARY=file
  RESULT_OBJECT_LIBRARY=file
  OBJECT_FILES=file
  DELETE_MODULES=list of name
  INCLUDE_FILE=file
  DELETE_OBJECT_FILES=boolean
  OUTPUT=file
  STATUS=status variable

Parameters **BASE_OBJECT_LIBRARY** or **BOL**

Specifies the name of the object library to use for the build.

The default parameter list assigns $BUILD_TARGET to this parameter. For information on using the default parameter list, see the remarks at the end of this processor description.

RESULT_OBJECT_LIBRARY or *ROL*

Specifies the name of the object library that results from the build.

By default, the name of result object library is the same as the name of the base object library except the cycle number of 1 higher.

OBJECT_FILES or *OF*

Specifies the names of the object files to use for the build.

The default is OBJECT_FILE.

DELETE_MODULES or *DM*

Specifies the names of the modules in the object library to delete during this procedure.

The default parameter list assigns $UNKNOWN_LIBRARY_ENTRIES to this parameter. For information on using the default parameter list, see the remarks at the end of this processor description.

INCLUDE_FILE or *IF*

Specifies a file of SCL commands to execute during this procedure.

DELETE_OBJECT_FILES or *DOF*

Specifies whether to delete the object files during this procedure.

The default is TRUE.

OUTPUT or *O*

Specifies a file to which messages are written during the execution of this procedure.

The default parameter list assigns $OUTPUT_FILE to this parameter. For information on using the default parameter list, see the remarks at the end of this processor description.

Remarks

- The Build Utility defines the following default parameter list for this processor:

  ```
  define_parameter_list..
    processor = build_object_library..
    parameter_list_name = default..
    parameter_list = 'bol=$build_target  o=$output_file'//..
                       dm=$unknown_library_entries'
  ```

  To use this default parameter list, include the following statement in your transformation file:

  ```
  include_command 'build_object_library '//..
                 $parameter_list_value(build_object_library default)
  ```

- This processor repeatedly tries to create cycle one of the result object library. This is done to insure the integrity of the object library in a multi-programming environment. If cycle one already exists, the build will never terminate.

EXPAND_SOURCE Build Utility Processor

Purpose Expands selected decks to a file.

Format **EXPAND_SOURCE** or
EXPS
DECKS=list of name or keyword
COMPILE=file
DEBUG_AIDS=keyword
OUTPUT_SOURCE_MAP=file
SELECTION_CRITERIA=file
WIDTH=integer
LINE_IDENTIFIER=keyword
BASE=file
ALTERNATE_BASES=list of file
LIST=file
ERRORS=file
EXPANSION_DEPTH=integer
DISPLAY_OPTIONS=keyword
ORDER=keyword
STATUS=status variable

Parameters *DECKS* or *DECK* or *D*

Specifies the decks to expand. Specify a deck name, a list of deck names, or one of the following keywords:

ALL

Specifies all decks.

NONE

If NONE is specified, SCU uses the file specified in the SELECTION_CRITERIA parameter to determine which decks to expand.

The default parameter list assigns $CHANGED_DECKS to this parameter. For information on using the default parameter list, see the remarks at the end of this processor description.

COMPILE or *C*

Specifies the name of a file to which the decks are expanded.

The default is COMPILE.

DEBUG_AIDS or *DA*

Specifies whether to write screen debug information to the file specified in the OUTPUT_SOURCE_MAP parameter.

Specify one of the following keywords:

DT

Debug information is written.

NONE

No debug information is written.

The default is NONE.

OUTPUT_SOURCE_MAP or *OSM*

Specifies a file to which the debug information is written.

The default is OUTPUT_SOURCE_MAP.

SELECTION_CRITERIA or *SC*

Specifies a file containing SCU selection commands.

By default, the DECKS parameter specifies the decks to expand.

WIDTH or *W*

Specifies the length of the expanded lines excluding the line identifiers.

By default, SCU uses the default line width from the deck header of each deck.

LINE_IDENTIFIER or *LI*

Specifies the placement of the SCU line identifiers.

Specify one of the following keywords:

> *RIGHT* or *R*
>
> Line identifiers are placed to the right of the text.
>
> *LEFT* or *L*
>
> Line identifiers are placed to the left of the text.
>
> *NONE*
>
> No line identifiers are placed.

By default, SCU uses the default line identifier placement specified in the deck header of each deck.

BASE or **B**

Specifies the file to use as the base source library.

This is a required parameter.

The default parameter list assigns $BASE_SOURCE_LIBRARY to this parameter. For information on using the default parameter list, see the remarks at the end of this processor description.

ALTERNATE_BASES or *ALTERNATE_BASE* or *AB*

Specifies one or more files to use as alternate source libraries.

By default, no alternative source libraries are used.

LIST or *L*

Specifies a file to which messages are written during this procedure.

The default parameter list assigns $OUTPUT_FILE to this parameter. For information on using the default parameter list, see the remarks at the end of this processor description.

ERRORS or *E*

Specifies a file to which error messages are written during this procedure.

The default parameter list assigns $ERRORS_FILE to this parameter. For information on using the default parameter list, see the remarks at the end of this processor description.

EXPANSION_DEPTH or *ED*

Specifies the number of levels of *COPY and *COPYC directives to process. Specify an integer in the range 0..262143.

By default, all *COPY and *COPYC directives are processed.

DISPLAY_OPTIONS or *DO*

Specifies whether the listing includes the library origin for each deck when more than one library is used. Specify one of the following keywords:

BRIEF or *B*

Does not list the library origins for each deck.

FULL or *F*

Lists the library origins.

The default is BRIEF.

ORDER or *O*

Specifies the order in which decks are expanded. Specify one of the following keywords:

COMMAND or *C*

Expands decks in the order specified on the DECK parameter.

LIBRARY or *L*

Expands decks in alphabetical order.

The default is LIBRARY.

Remarks The Build Utility defines the following default parameter list for this processor:

```
define_parameter_list..
  processor = expand_source..
  parameter_list_name = default..
  parameter_list = 'd=$changed_decks b=$base_source_library..
                    ab=$alternate_source_libraries l=$output_file..
                    e=errors_file'
```

To use this default parameter list, include the following statement in your transformation file:

```
include_command 'expand_source '//..
                 $parameter_list_value(expand_source default)
```

PPE Subcommands

This section contains descriptions of two PPE subcommands. These commands are provided because they are used to start the Build Utility from PPE.

Each description provides the following information:

- Purpose of the command.
- Format of the command.
- Descriptions of the command parameters.

Optionally, the command descriptions may contain remarks and examples of command usage.

The command format provides the full command name, any abbreviations of the command, and the parameters for the command.

The STATUS parameter on the Build Utility commands is the same STATUS parameter available on all NOS/VE commands. If the STATUS parameter is specified, it must be specified by name, not by position.

BUILD_CHANGED_DECKS PPE Subcommand

Purpose Builds all expandable decks that have changed since the last build.

Format **BUILD_CHANGED_DECKS** or
BUICD or
BUILD
BUILD_TARGET=list of file or keyword
EXECUTE_TRANSFORMATIONS=boolean
OUTPUT=file
ERRORS=file
DISPLAY_OPTIONS=keyword
PERFORM_BUILD=keyword
STATUS=status variable

Parameters *BUILD_TARGET* or *BT*

Specifies the build targets to analyze. You can reference the build targets by name, or specify the following keyword:

ALL

Specifies all build targets in the system.

By default, the Build Utility analyzes only the first build target defined in the PPE-generated input.

EXECUTE_TRANSFORMATIONS or *ET*

Specifies whether to execute the transformations for an out-of-date build target.

This parameter accepts a boolean value. If you specify FALSE, the transformations are not made. Only the analysis phase of the build is executed. This allows you to determine which decks would be built without actually performing a build.

By specifying the DISPLAY_OPTIONS parameter, you can cause the Build Utility to display the out-of-date build targets and the reasons they were found to be out of date.

The default value is TRUE.

OUTPUT or *O*

Specifies the name of the file to contain the information generated by the DISPLAY_OPTIONS parameter.

The default value is $OUTPUT.

ERRORS or *E*

Specifies the name of the file to contain the error messages.

The default value is $ERRORS.

DISPLAY_OPTIONS or *DO*

Specifies the information to display about the build. The Build Utility writes this information to the file specified by the OUTPUT parameter. Specify one of the following keywords:

ANALYSIS_TRACE or AT

Writes the steps taken by the Build Utility during the build.

ANALYSIS_RESULTS or AR

Writes the results of the build. These results include the build targets and decks that were out of date.

NONE

Indicates that no information is written.

The default is NONE.

PERFORM_BUILD or *BP*

Specifies whether to execute the PPE-generated input to the Build Utility in batch or interactive mode. Specify one of the following keywords:

INTERACTIVELY or *I*

Runs the Build Utility interactively.

BATCH or *B*

Runs the Build Utility in batch mode.

The default is INTERACTIVELY.

Remarks This command can only be entered from the home line in PPE.

BUILD_DECKS PPE Subcommand

Purpose Builds each expandable deck selected by the user.

Format **BUILD_DECKS** or
BUID
BUILD_TARGET=list of file or keyword
DECKS=name or keyword or list of name
OUTPUT=file
ERRORS=file
DISPLAY_OPTIONS=keyword
PERFORM_BUILD=keyword
STATUS=status variable

Parameters *BUILD_TARGET* or *BT*

Specifies the build targets to analyze. You can reference the build targets by name, or specify the following keyword:

ALL

Specifies all build targets in the system.

By default, the Build Utility analyzes only the first build target defined in the PPE-generated input.

DECKS or *D*

Specifies the decks to build. You can specify a deck name, a list of names, or the keyword ALL.

ALL

Specifies all decks. This is the same as a full build.

There are only three occasions when you will want to specify this parameter.

- When you want a full build.
- When you know that decks were changed in such a way that the object code will not be affected. For example, changing a comment in a source deck.
- When you know exactly which decks need to be built.

By default, the Build Utility determines which decks are out of date by comparing the date/time stamp on the source deck with the date/time stamp on the build target.

OUTPUT or *O*

Specifies the name of the file to contain the information generated by the DISPLAY_OPTIONS parameter.

The default value is $OUTPUT.

ERRORS or *E*

Specifies the name of the file to contain the error messages.

The default value is $ERRORS.

DISPLAY_OPTIONS or *DO*

Specifies the information to display about the build. The Build Utility writes this information to the file specified by the OUTPUT parameter. Specify one of the following keywords:

ANALYSIS_TRACE or AT

Writes the steps taken by the Build Utility during the build.

ANALYSIS_RESULTS or AR

Writes the results of the build. These results include the build targets and decks that were out of date.

NONE

Indicates that no information is written.

The default is NONE.

PERFORM_BUILD or *BP*

Specifies whether to execute the PPE-generated input to the Build Utility in batch or interactive mode. Specify one of the following keywords:

INTERACTIVELY or *I*

Runs the Build Utility interactively.

BATCH or *B*

Runs the Build Utility in batch mode.

The default is INTERACTIVELY.

Remarks This command can only be entered from the home line in PPE.

Build Utility Examples 6

Build Utility Examples 6

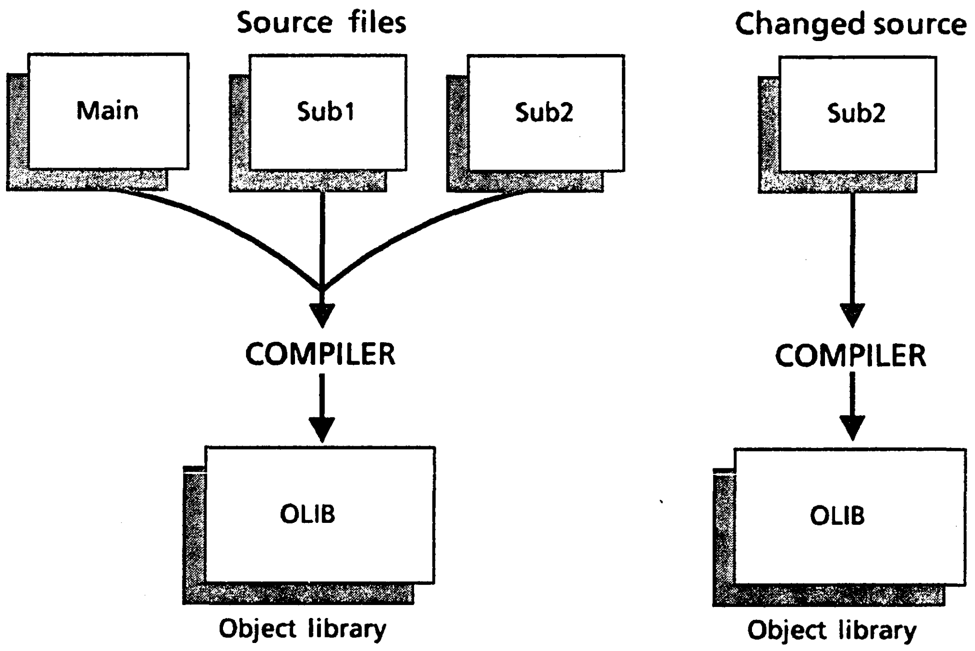
This chapter provides examples of using the Build Utility. These are interactive examples because you are expected to enter them from your terminal as you read through them. The first example uses the Build Utility to build and update a simple file system. The second example uses the Build Utility to build and update a simple library system. The last example uses the Build Utility on a system with two build targets. All files in these examples must be created as permanent files, not created in $LOCAL.

A File System Example

In the following example, you create a file system and run the Build Utility to update the system.

What it is supposed to do

This example shows how to create a small FORTRAN program that consists of three program modules: the main program and two subroutines. Each program module is stored in a separate file. Use the Build Utility to create an executable object library from these source files. Then, make a change to one of the source files and run the Build Utility to update the object library. All files in this example must be created as permanent files.



1. Create the following three files. Each file is a module of a FORTRAN program. Remember, to follow FORTRAN format, you must start each line in column 7.

   File name: `main`

   Contents:

   ```
   PROGRAM MAIN
   PRINT *,'Enter a number:'
   READ (*,*) X
   CALL SUB1 (X,Y)
   CALL SUB2 (Y,Z)
   PRINT *,Z
   END
   ```

   File name: `sub1`

   Contents:

   ```
   SUBROUTINE SUB1 (X,Y)
   Y=X+1.0
   RETURN
   END
   ```

   File name: `sub2`

   Contents:

   ```
   SUBROUTINE SUB2 (Y,Z)
   Z=Y*10.0
   RETURN
   END
   ```

2. Create the following Build Utility input file. This file defines one build target that dependent on the three program modules.

   File name: `input1`

   Contents:

   ```
   define_build_target ..
     build_target=olib ..
     build_target_kind=none ..
     dependences=(main sub1 sub2) ..
     transformation=tfile
   ```

3. Create the following transformation file. This file specifies the transformation that the Build Utility uses to create the build target.

   File name: `tfile`

   Contents:

   ```
   changed=$dependences
   fortran i=changed b=object_file
   build_object_library ..
     base_object_library=olib ..
     object_files=object_file
   ```

4. Run the Build Utility by entering the following command:

   ```
   /build_software input1 display_options=(analysis_trace analysis_results)
   ```

   or abbreviated,

   ```
   /buis input1 do=(at ar)
   ```

   NOTE

   If you get error messages while the Build Utility is running, verify that you entered the above information correctly. If you still cannot find the mistake, look for the message in appendix D, Diagnostic Messages.

5. Verify that the object library (OLIB) was created by entering the following command:

   ```
   /display_object_library olib
   ```

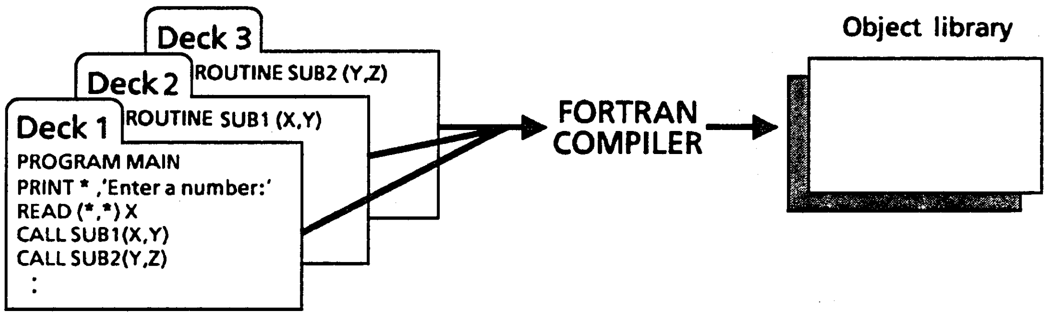
6. You can look at the CHANGED file to see the files the Build Utility compiled to create the object file.
7. Edit file SUB2 to change the statement Z=Y*10.0 to Z=Y*30.0.
8. Run the Build Utility by repeating step 4.
9. Look at the CHANGED file and notice that only SUB2 was recompiled. (You can use the COPY_FILE command to look at SUB2.)

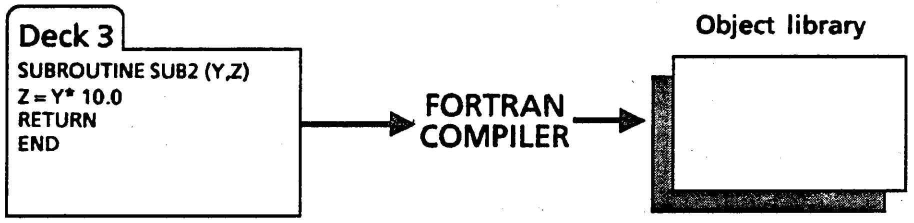
A Library Example

In the following example, you create a source library and use the Build Utility to update the library.

What it is supposed to do

This example creates a source library, containing three decks. Each deck is a module of a FORTRAN program. You use the Build Utility to build an object library from your source library. Then, change one of the decks in the source library and use the Build Utility to update the object library.





1. Start the Source Code Utility by entering the following command:

```
/scu
```

2. Create a source library named SLIB by entering the following command:

```
sc/create_library slib
```

3. Use the EDIT_DECK command to create the following SCU decks. Each deck is a module of a FORTRAN program. Remember to start each line in column 7. (If you worked through the previous example, you can use the CREATE_DECK command to convert the source from that example to decks.) The following example uses the EDIT_DECK command.

```
sc/edit_deck main m=kw321
```

Deck name: `main`

Contents:

```
PROGRAM MAIN
PRINT *,'Enter a number:'
READ (*,*) X
CALL SUB1 (X,Y)
CALL SUB2 (Y,Z)
PRINT *,Z
END
```

Deck name: `sub1`

Contents:

```
SUBROUTINE SUB1 (X,Y)
Y=X+1.0
RETURN
END
```

Deck name: `sub2`

Contents:

```
SUBROUTINE SUB2 (Y,Z)
Z=Y*10.0
RETURN
END
```

4. Quit the Source Code Utility by entering the following:

```
sc/quit
```

5. Create the following Build Utility input file:

   File name: `input2`

   Contents:

```
define_processor..
   processor=fortran

define_parameter_list..
   processor=fortran..
   parameter_list='i=compile b=object_file'

define_source_libraries..
   internal_source_libraries=slib..
   default_processor=fortran

define_build_target..
   build_target=olib..
   build_target_kind=object_library..
   composition='include_deck deck=(main sub1 sub2)'..
   transformation=default
```

6. Start the Build Utility by entering the following command:

```
/build_software input2 display_options=(analysis_trace analysis_results)
```

   or abbreviated,

```
/buis input2 do=(at ar)
```

   NOTE

   If you get error messages while the Build Utility is running, verify that you entered the above information correctly. If you still cannot find the mistake, look for the message in appendix D, Diagnostic Messages.

7. Verify that the object library (OLIB) was created by entering the following command:

```
/display_object_library olib
```

8. Start SCU and edit deck SUB2 to change the statement Z=Y*10.0 to Z=Y*30.0. (You must enter USE_LIBRARY SLIB first.)

9. Quit SCU.

10. Run the Build Utility by repeating step 6.

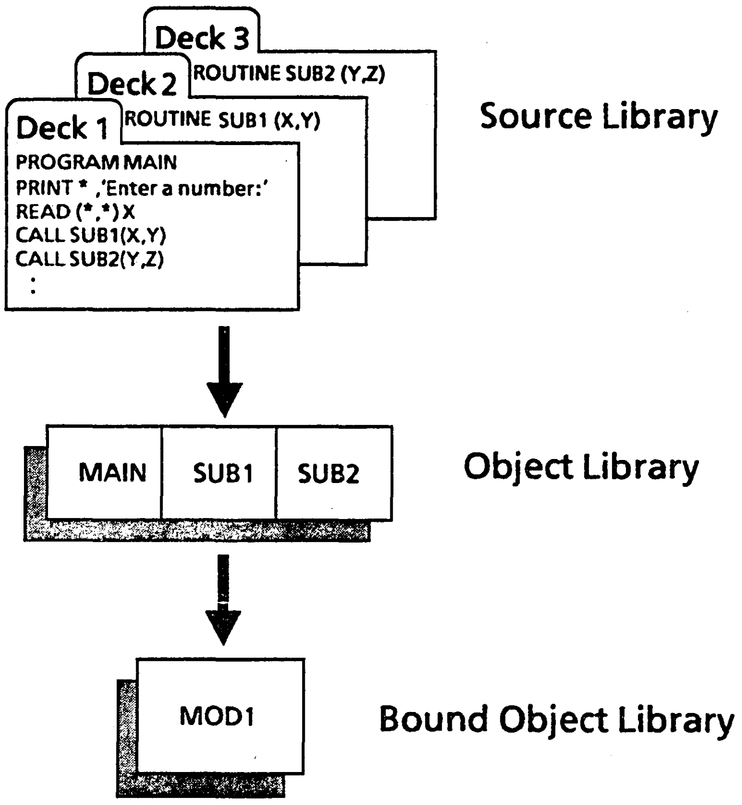
11. You can use the COPY_FILE command to look at the COMPILE file and see that only SUB2 was recompiled.

Multiple Build Target Example

In the following example, you create a library system and use the Build Utility to perform more complex transformations.

What it is supposed to do

This system has two build targets. First, the source library is compiled into an object library. Then, the object library is converted to a bound module.



1. Use the source library you created in the previous library example. If you do not have this library, follow steps 1 through 5 from the Library Example section of this chapter.

2. Create the following Build Utility input file:

   File name: `input3`

   Contents:

```
define_processor ..
   processor=fortran

define_parameter_list ..
   processor=fortran ..
   parameter_list='i=compile b=object_file'

define_source_libraries ..
   internal_source_libraries=slib ..
   default_processor=fortran

define_build_target ..
   build_target=blib ..
   build_target_kind=none ..
   dependences=olib ..
   transformation=transfile

define_build_target ..
   build_target=olib ..
   build_target_kind=object_library ..
   composition='include_deck d=(main sub1 sub2)' ..
   transformation=default
```

3. Create the following transformation file:

   File name: `transfile`

   Contents:

```
create_object_library
create_module name=mod1 component=olib
generate_library blib
quit
```

4. Run the Build Utility by entering the following command:

```
/build_software input3 display_options=(analysis_trace analysis_results)
```

   or abbreviated,

```
/buis input3 do=(at ar)
```

5. Notice from the display options that the first build target was analyzed and found to be out of date, or, in this case, non-existent. Therefore, the transformation was performed and the build target was created. This process was repeated for the second build target.

6. Edit deck SUB2 to change the statement Z=Y*10.0 to Z=Y*30.0.

7. Use the COPY_FILE command to look at the COMPILE file and see that only SUB2 was recompiled.

Appendixes

Glossary A

This appendix lists terms and their definitions as used in this manual.

B

Build

The process by which the Build Utility constructs build targets. This process involves executing a transformation on the files that compose the build target.

Build Processor

The software that the Build Utility uses during a transformation.

Build Target

A file that is the result of a transformation of the files or decks that it depends on.

C

Compilation

The process of transforming source text into executable object code.

Composition

A list of source decks that together compose the build target.

D

Data Dependency

The relationship between the input and output of a transformation. The contents of the output are dependent on the contents of the input. For example, the file produced by binding an object library is dependent on the object library.

Deck

An SCU object consisting of text with a descriptive header. A deck is a subset of an SCU library.

Default Parameter List

The currently selected parameter list for the build processor.

Dependency Graph

A visual representation of data dependency between files.

E

Expansion

The process by which SCU converts the text in a deck to a file that can be used as input to a build processor.

F

File

A collection of information referenced by a unique name. It is the smallest unit of information that can be directly referenced by a NOS/VE command.

File System

A group of related files. In this manual, these files are used to construct a build target.

H

Home Line

The line on the terminal screen to which the cursor moves when the HOME key is pressed. The user can enter commands from the home line.

I

Input File

A file that contains the information needed by the Build Utility. It is a required parameter for the BUILD_SOFTWARE command.

L

Layers

Storage method that uses several levels of libraries. The Build Utility searches the layers in the order given to find the most current version of source text.

O

Object Library

A file containing one or more executable modules and a directory to each module and its entry points.

Object Module

A compiler-generated unit containing object code and instructions for loading the object code. Object modules are stored in object libraries.

P

Parameter List

A string of specifications passed to a task when its execution begins. The Build Utility passes a parameter list to each build processor.

Preprocessor

A processor that prepares the source text for the main processor. The Build Utility recognizes two preprocessors: DMCPC and DMFPC. These are the IM/DM preprocessors for COBOL and FORTRAN, respectively.

Processor

A NOS/VE command which is used during a transformation. For example, a compiler is a processor that transforms source text to object code.

S

Source Library

A collection of decks on a file with a descriptive header, generated and maintained by SCU.

Source Text

A collection of related text as it is provided to the system by the user. This text can be stored in files or library decks.

T

Transformation

The actions that are required to create a build target.

The NOS/VE User Manual Set B

This manual is part of a set of user manuals that describe NOS/VE and NOS/VE applications. The descriptions of these manuals follow:

Introduction to NOS/VE

Introduces NOS/VE and SCL to users who have no previous experience with them. It describes, in tutorial style, the basic concepts of NOS/VE: creating and using files and catalogs of files, executing and debugging programs, submitting jobs, and getting help online.

The manual describes the conventions followed by all NOS/VE commands and parameters, and lists many of the major commands, products, and utilities available on NOS/VE.

NOS/VE System Usage

Describes the command interface to NOS/VE using the SCL language. It describes the complete SCL language specification, including language elements, expressions, variables, command stream structuring, and procedure creation. It also describes system access, interactive processing, access to online documentation, file and catalog management, job management, tape management, and terminal attributes.

NOS/VE File Editor

Describes the EDIT_FILE utility used to edit NOS/VE files and decks. The manual has basic and advanced chapters describing common uses of the utility, including creating files, copying lines, moving text, editing more than one file at a time, and creating editor procedures. It also contains descriptions of subcommands, functions, and terminals.

NOS/VE Source Code Management

Describes the SOURCE_CODE_UTILITY, a development tool used to organize and maintain libraries of ASCII source code. Topics include deck editing and extraction, conditional text expansion, modification state constraints, and using the EDIT_FILE utility.

NOS/VE Object Code Management

Describes the CREATE_OBJECT_LIBRARY utility used to store and manipulate units of object code within NOS/VE. Program execution is described in detail. Topics include loading a program, program attributes, object files and modules, message module capabilities, code sharing, segment types and binding, ring attributes, and performance options for loading and executing.

NOS/VE Advanced File Management

Describes three file management tools: Sort/Merge, File Management Utility (FMU), and keyed-file utilities. Sort/Merge sorts and merges records; FMU reformats record data; and the keyed-file utilities copy, display, and create keyed files (such as indexed-sequential files).

NOS/VE Terminal Definition

Describes the DEFINE_TERMINAL command and the statements that define terminals for use with full-screen applications (for example, the EDIT_FILE utility).

NOS/VE Commands and Functions

Lists the formats of the commands, functions, and statements described in the NOS/VE user manual set. A format description includes brief explanations of the parameters and an example using the command, function, or statement.

Professional Programming Environment

Describes the Professional Programming Environment (PPE), a full-screen development tool for coordinating multi-person programming objects.

The manual introduces PPE concepts, describes how get started using PPE, and provides procedures for performing PPE tasks. It also contains individual descriptions of each PPE screen and PPE command, presented alphabetically.

Ordering Printed Manuals

You can order printed Control Data manuals from your local Control Data sales office. Sites in the U.S. can also order manuals from the following address:

Control Data Corporation
Literature and Distribution Services
308 North Dale Street
St. Paul, Minnesota 55103

When ordering manuals, please indicate whether you require the entire manual or just the latest revision packet.

Diagnostic Messages C

This section lists the Build Utility diagnostic messages. It includes a description of each message and a suggested user action.

--CATASTROPHIC BU 1-- The Build Utility cannot continue because there is not enough space in the segment reserved for analysis results to hold all of the output from the analysis of build target {file}.

Condition Identifier: BUE$ANALYSIS_AREA_FULL

Description: Your build system has exceeded the memory segment assigned to the analysis results.

User Action: Decrease the size of your build system.

--ERROR BU 4-- Build target {file} was declared to be an object library. However, the file you specified is not in the correct format to be an object library.

Condition Identifier: BUE$BAD_OBJECT_LIBRARY

Description: You have specified that a build target is an object library. However, the file you specified is not in the correct format to be an object library.

User Action: Delete the current file specified as an object library. The Build Utility will create a new file which is in the correct format.

--ERROR BU 10-- The definition of build target {file} must specify the COMPOSITION_MAP parameter when MAPPED_DECKS_ONLY is given as the value of the COMPOSITION parameter.

Condition Identifier: BUE$COMPOSITION_MAP_REQUIRED

Description: You must specify a composition map when the COMPOSITION parameter of the DEFINE_BUILD_TARGET command is set to MAPPED_DECKS_ONLY.

User Action: Specify a composition map.

--ERROR BU 11-- The deck {text} specified with the DECKS parameter of BUILD_SOFTWARE appears in the composition of build target {file} but no deck by that name exists.

Condition Identifier: BUE$DECK_NONEXISTENT

Description: You have specified that a particular deck should be built. Although this deck appears in the composition of the build target, it does not exist in the source library.

User Action: Verify that the name of the deck requested matches the name of the deck in the source library.

--CATASTROPHIC BU 12-- The Build Utility cannot continue because there is not enough space in the segment reserved for definitions to hold all of the definitions created by subcommands in the INPUT file.

Condition Identifier: BUE$DEFINITIONS_AREA_FULL

Description: Your build exceeded the limits of the memory segment assigned to hold the definitions of your system.

User Action: Decrease the size of your definition section in the input file.

--ERROR BU 14-- The name {text} must not be defined as a parameter list for processor {text} more than once.

Condition Identifier: BUE$DUPLICATE_PLIST_DEFINITION

Description: You cannot have more than one DEFINE_PARAMETER_LIST for a specified processor with the same PARAMETER_LIST_NAME.

User Action: Remove duplicate occurrences of all parameter list names. If a parameter list has different attributes, assign it a different name.

--ERROR BU 15-- The name {text} must not be defined as a processor more than once.

Condition Identifier: BUE$DUPLICATE_PROC_DEFINITION

Description: The same processor name cannot be defined more than once in your input file.

User Action: Remove duplicate occurrences of all processor definitions.

--ERROR BU 16-- There must not be more than one DEFINE_SOURCE_LIBRARIES command in the utility INPUT file.

Condition Identifier: BUE$DUPLICATE_SOURCE_DEFINITION

Description: You cannot have more than one occurrence of the DEFINE_SOURCE_LIBRARIES command in your input file.

User Action: Remove duplicate occurrences of the DEFINE_SOURCE_LIBRARIES command from your input file.

--ERROR BU 17-- The file {file} must not be defined as a build target more than once.

Condition Identifier: BUE$DUPLICATE_TARGET_DEFINITION

Description: You cannot define any given file as a build target more than once in an input file.

User Action: Remove duplicate occurrences of the DEFINE_BUILD_TARGET command that contain the same file name.

--ERROR BU 21-- The definition of build target {file} specifies that it is of kind {text} and also specifies the LAYERS parameter — that parameter may only be specified for build targets of kind OBJECT_LIBRARY.

Condition Identifier: BUE$LAYERS_USED_FOR_NON_OBJLIB

Description: The LAYERS parameter can be specified for a build target that is defined with a BUILD_TARGET_KIND of OBJECT_LIBRARY.

User Action: You must either remove the LAYERS parameter from the DEFINE_BUILD_TARGET command or change the BUILD_TARGET_KIND to OBJECT_LIBRARY.

--ERROR BU 22-- The definition of build target {file} specifies that it is of kind {text} and that the default transformation should be used — the utility provides a default transformation only for build targets of kind OBJECT_LIBRARY.

Condition Identifier: BUE$NO_DEFAULT_TRANSFORMATION

Description: The Build Utility provides a default transformation only for build targets of kind OBJECT_LIBRARY. If you specify the BUILD_TARGET_KIND as NONE, you must provide your own transformations.

User Action: If the build target is not an object library, change the TRANSFORMATION parameter to specify a file that contains the SCL commands to use in the transformation.

--ERROR BU 23-- No work can be done by the utility because the INPUT file did not contain a subcommand that defined a build target.

Condition Identifier: BUE$NO_TARGETS_DEFINED

Description: An input file must contain at least one build target definition. If you do not specify a build target, the Build Utility has nothing to process. This message is always generated when you quit a session of the Build Utility without defining a build target.

User Action: Define a build target in your input file using the DEFINE_BUILD_TARGET command.

--ERROR BU 27-- The name of the preprocessor for processor {text} must not be the same as the name of the processor itself.

Condition Identifier: BUE$PROC_AND_PREPROC_SAME

Description: The name specified by the PREPROCESSOR parameter is the same as the name specified by the PROCESSOR parameter.

User Action: Change or eliminate the name of the preprocessor.

--CATASTROPHIC BU 29-- The Build Utility cannot continue because there is not enough space in the target stack for the build targets that have not yet been analyzed.

Condition Identifier: BUE$TARGET_STACK_FULL

Description: The memory space available for the build has been exceeded.

User Action: Decrease the number of build targets in your system.

--ERROR BU 31-- The processor {text} must be defined before a parameter list for it may be defined.

Condition Identifier: BUE$UNDEFINED_PROCESSOR

Description: You must define a processor before you can define a parameter list to be used by that processor.

User Action: Put the DEFINE_PROCESSOR command before all associated DEFINE_PARAMETER_LIST commands in the input file.

--ERROR BU 32-- The BUILD_TARGET parameter specified file {file}, but that file was not defined as a build target by utility subcommands in the INPUT file.

Condition Identifier: BUE$UNDEFINED_TARGET_SPECIFIED

Description: You cannot request the Build Utility to build a target that is not defined with a DEFINE_BUILD_TARGET command in the input file.

User Action: Add a DEFINE_BUILD_TARGET command to the input file for the build target.

--ERROR BU 33-- The definition of build target {file} must not specify the DEPENDENCES parameter when the BUILD_TARGET_KIND parameter is OBJECT_LIBRARY and the TRANSFORMATION parameter is DEFAULT.

Condition Identifier: BUE$DEPENDENCES_NOT_ALLOWED

Description: The DEPENDENCES parameter is used for build targets of kind NONE. If the BUILD_TARGET_KIND is OBJECT_LIBRARY, use the COMPOSITION parameter. Also, the TRANSFORMATION parameter cannot be DEFAULT if the DEPENDENCES parameter is used because there is no default transformation for files.

User Action: Remove the DEPENDENCES parameter from your build target definition, or change the target kind to NONE and specify a transformation file.

--ERROR BU 34-- The definition of build target {file} must specify the COMPOSITION parameter when the BUILD_TARGET_KIND parameter is OBJECT_LIBRARY and the TRANSFORMATION parameter is DEFAULT.

Condition Identifier: BUE$DEFAULT_TRANS_NEEDS_COMP

Description: The COMPOSITION parameter of the DEFINE_BUILD_TARGET command is a required parameter when the build target is of kind object library and the default transformation is requested.

User Action: Use the COMPOSITION parameter to specify the selection criteria for the decks that compose the object library.

--ERROR BU 35-- The processor for deck {text} cannot be determined because the processor attribute of the deck is empty and no default processor is defined.

Condition Identifier: BUE$CANT_DETERMINE_PROCESSOR

Description: No processor is specified in the deck header of the source deck and no default processor is defined in the DEFINE_SOURCE_LIBRARIES command. Therefore, the Build Utility cannot determine which processor to use.

User Action: Assign a default processor using the DEFAULT_SOURCE_LIBRARIES command.

--ERROR BU 36-- The processor attribute for deck {text} specifies the processor {text}, but that processor cannot be used because it has not been defined to the utility via the DEFINE_PROCESSOR subcommand.

Condition Identifier: BUE$CANT_USE_PROCESSOR

Description: The processor attribute of the deck header specifies a processor that has not been defined using the DEFINE_PROCESSOR command or defined on the PPE build processor screen.

User Action: Define the processor or change the processor attribute of the deck header.

--ERROR BU 38-- The parameter list {text} for processor {text} is needed to build deck {text}, but it is not defined.

Condition Identifier: BUE$UNDEFINED_PARAMETER_LIST

Description: Every processor must have at least one parameter list defined. One of your processors does not.

User Action: Define at least one parameter list for each processor using the DEFINE_PARAMETER_LIST command.

--ERROR BU 39-- Analysis halted at build target {file} because its transformation terminated abnormally.

Condition Identifier: BUE$TRANSFORMATION_FAILED

Description: The build failed. The transformation for one of the build targets terminated abnormally causing the build analysis to halt.

User Action: Verify the transformation procedure for the suspect build target.

--WARNING BU 40-- The processor attribute for deck {text} specifies that the {text} parameter list for processor {text} should be used, but since that list is not defined, the default list {text} is being used.

Condition Identifier: BUE$USING_DEFAULT_PLIST

Description: The processor attribute of the deck header specifies a parameter list to be used. Since this parameter list is not defined, the Build Utility used the default parameter list for that processor instead.

User Action: Not applicable.

Build Utility Concepts D

This appendix defines files, decks, libraries and file systems and identifies the differences between them. It also discusses the concepts of data dependency and transformations.

Files Versus Decks

The standard definition of a file is *a collection of information which is referenced by a unique name*. However, this broad definition can apply to a deck as well. Because both decks and files are units of information processed by the Build Utility, it is important to understand the distinction between them.

File

A file is a collection of information which is referenced by a unique name. It is the smallest unit of information that can be directly referenced by a NOS/VE command.

Deck

Although a deck is a collect of text which is referenced by a unique name, it is not a file. A deck is a subset of one type of file, a library. Therefore, a deck cannot be directly referenced without referencing the library that contains that deck.

Using a File System

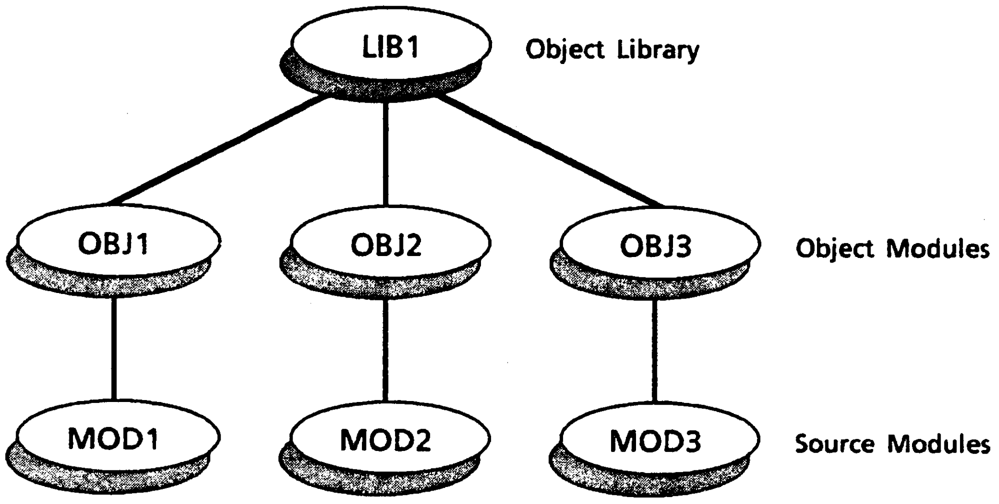
The Build Utility is able to handle file systems and libraries. A file system is a group of related files. The content of the files can be any collection of ASCII text. For an example of a file system, consider a documentor writing a manual. Writing each chapter in a separate file is convenient because you can update sections without having to access the entire system. These separate files are eventually combined into a single file, the book or build target.

Using a Source Library

A source library is a special type of file. A library can be referenced directly from the NOS/VE command line. A source library is divided into decks. These decks cannot be referenced directly without referencing the library first. There are two ways to create a library in NOS/VE. You can use the SOURCE_CODE_UTILITY subcommand CREATE_LIBRARY. You can also use PPE to create a source library.

Data Dependency

For the Build Utility, data dependency refers to the dependence of build targets on the files that compose them. For example, dependency exists between source modules and object code. If you create an object library from several source modules, the contents of the object code is dependent on the source code. A dependency graph is a useful tool for visually describing data dependency. The following diagram is a dependency graph for a simple library system.



From the graph, you can see that library LIB1 is dependent on modules OBJ1, OBJ2, and OBJ3. These modules are, in turn, dependent on MOD1, MOD2, and MOD3, respectively.

Transformations

A transformation is one or more NOS/VE commands that manipulate data. In the example above, it is a transformation that creates object modules from source modules. Although this transformation is a compiler, there are many transformations that are not compilers. For example, expanding a deck is a transformation. The Build Utility allows you to create your own transformation by specifying a transformation file.

Index

Comments (continued from other side)

ʾlease fold on dotted line;
ˌeal edges with tape only.

FOLD

ꟳOLD

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
First-Class Mail Permit No. 8241 Minneapolis, MN

POSTAGE WILL BE PAID BY ADDRESSEE

CONTROL DATA
Technical Publications
SVL104
P.O. Box 3492
Sunnyvale, CA 94088-3492

The Build Utility **60487413 A**

We would like your comments on this manual to help us improve it. Please take a few minutes to fill out this form.

Who are you?	**How do you use this manual?**
☐ Manager	☐ As an overview
☐ Systems analyst or programmer	☐ To learn the product or system
☐ Applications programmer	☐ For comprehensive reference
☐ Operator	☐ For quick look-up
☐ Other ______	☐ Other ______

What programming languages do you use? ______

How do you like this manual? Answer the questions that apply.

Yes	Somewhat	No	
☐	☐	☐	Does it tell you what you need to know about the topic?
☐	☐	☐	Is the technical information accurate?
☐	☐	☐	Is it easy to understand?
☐	☐	☐	Is the order of topics logical?
☐	☐	☐	Can you easily find what you want?
☐	☐	☐	Are there enough examples?
☐	☐	☐	Are the examples helpful? (☐ Too simple? ☐ Too complex?)
☐	☐	☐	Do the illustrations help you?
☐	☐	☐	Is the manual easy to read (print size, page layout, and so on)?
☐	☐	☐	Do you use this manual frequently?

Comments? If applicable, note page and paragraph. Use other side if needed.

Check here if you want a reply: ☐

Name	Company
Address	Date
	Phone

Please send program listing and output if applicable to your comment.

CONTROL DATA